Introducing Computational Thinking in K-12 Education: Historical, Epistemological, Pedagogical, Cognitive, and Affective Aspects

Michael Lodi

Alma Mater Studiorum · Università di Bologna

PhD program in
COMPUTER SCIENCE AND ENGINEERING

INTRODUCING COMPUTATIONAL THINKING IN K-12 EDUCATION: HISTORICAL, EPISTEMOLOGICAL, PEDAGOGICAL, COGNITIVE, AND AFFECTIVE ASPECTS

Presented by:
MICHAEL LODI
Dep. of Computer Science and Engineering
INRIA Focus team

PhD program coordinator:
Prof. DAVIDE SANGIORGI

Supervisor:
Prof.SIMONE MARTINI

DOI: 10.6092/unibo/amsdottorato/9188
http://amsdottorato.unibo.it/9188/

Contents

Abstract

Introduction of scientific and cultural aspects of Computer Science (CS) (called "Computational Thinking" - CT) in K-12 education is fundamental. We focus on three crucial areas.

- **Historical, philosophical, and pedagogical aspects.** What are the big ideas of CS we must teach? What are the historical and pedagogical contexts in which CT emerged, and why are relevant? What is the relationship between learning theories (e.g., constructivism) and teaching approaches (e.g., plugged and unplugged)?
- **Cognitive aspects.** What is the sentiment of generalist teachers not trained to teach CS? What misconceptions do they hold about concepts like CT and "coding"?
- **Affective and motivational aspects.** What is the impact of personal beliefs about intelligence (mindset) and about CS ability? What the role of teaching approaches?

This research has been conducted both through historical and philosophical argumentation, and through quantitative and qualitative studies (both on nationwide samples and small significant ones), in particular through the lens of (often exaggerated) claims about transfer from CS to other skills.

Four important claims are substantiated.

- ✓ CS should be introduced in K-12 as a tool to understand and act in our digital world, and to use the power of computation for meaningful learning. CT is the conceptual sediment of that learning. We designed a curriculum proposal in this direction.
- ✓ Expressions "computational thinking" (useful to distantiate from digital literacy) and "coding" can cause misconceptions among teachers, who focus mainly on transfer to general thinking skills. Both disciplinary and pedagogical teacher training is hence needed.
- ✓ Some plugged and unplugged teaching tools have intrinsic constructivist characteristics that can facilitate CS learning, as shown with proposed activities.
- ✓ Growth mindset is not automatically fostered by CS, while not studying CS can foster fixed beliefs. Growth mindset can be fostered by creative computing, leveraging on its constructivist aspects.

Publications by the Author

During my PhD, I published, with different co-authors or alone, some research publications, listed below.

Publications in bold are part of this dissertation.

Journal Articles

- Lodi [2014b]. Learn computational thinking, learn to program. *Mondo Digitale* 13, 51 (2014), 822–833. *Article extracted from my Master degree dissertation [Lodi, 2014a], about computational thinking and difficulties in learning to program. In Italian.*
- **Lodi, Martini, and Nardelli [2017]. Do we really need computational thinking? Mondo Digitale 72, Article 2 (2017), 15 pages.** *In Italian.*
- **Bell and Lodi [2019a]. Constructing Computational Thinking Without Using Computers. Constructivist Foundations 14, 3 (2019), 342–351.**
- **Bell and Lodi [2019b]. Authors' Response: Keeping the "Computation" in "Computational Thinking" Through Unplugged Activities. Constructivist Foundations 14, 3 (2019), 357–359.**
- **Lodi, Malchiodi, Monga, Morpurgo, and Spieler [2019]. Constructionist Attempts at Supporting the Learning of Computer Programming: A Survey. Olympiads in Informatics 13 (2019), 99–121.**

Conference Proceedings

- **Corradini, Lodi, and Nardelli [2017a]. Computational Thinking in Italian Schools: Quantitative Data and Teachers' Sentiment Analysis after Two Years of "Pro-gramma Il Futuro". In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (Bologna, Italy) (ITiCSE '17). ACM, New York, NY, USA, 224–229.**
- **Corradini, Lodi, and Nardelli [2017b]. Conceptions and Misconceptions about Computational Thinking among Italian Primary School Teachers. In Proceedings of the 2017 ACM Conference on International Computing Education Research (Tacoma, Washington, USA) (ICER '17). ACM, New York, NY, USA, 136–144.**

- **Lodi [2018c]. Pensiero Computazionale: dalle "scuole di samba della computazione" ai CoderDojo. In Atti del convegno DIDAMATICA 2018. AICA, Cesena, Italy.** *In Italian.*

- **Monga, Lodi, Malchiodi, Morpurgo, and Spieler [2018]. Learning to program in a constructionist way. In Proceedings of Constructionism 2018: Constructionism, Computational thinking and Educational Innovation (Vilnius, Lithuania). 901–924.**

- **Lodi [2018a]. Can Creative Computing Foster Growth Mindset? In Proceedings of the 1st Systems of Assessments for Computational Thinking Learning workshop (TACKLE 2018) co-located with 13th European Conference on Technology Enhanced Learning (ECTEL 2018), Leeds, United Kingdom, September 3rd, 2018. CEUR Workshop Proceedings, Vol. 2190.**

- **Corradini, Lodi, and Nardelli [2018b]. An Investigation of Italian Primary School Teachers' View on Coding and Programming. In Informatics in Schools. Fundamentals of Computer Science and Software Engineering. Lecture Notes in Computer Science (ISSEP 2018), Sergei N. Pozdniakov and Valentina Dagienė (Eds.), Vol. 11169. Springer International Publishing, Cham, 228–243.**

- **Forlizzi, Lodi, Lonati, Mirolo, Monga, Montresor, Morpurgo, and Nardelli [2018]. A Core Informatics Curriculum for Italian Compulsory Education.In Informatics in Schools. Fundamentals of Computer Science and Software Engineering. Lecture Notes in Computer Science (ISSEP 2018), Sergei N. Pozdniakov and Valentina Dagienė (Eds.), Vol. 11169. Springer International Publishing, Cham, 141–153.**

- Borchia, Carbonaro, Casadei, Forlizzi, Lodi, and Martini [2018]. Problem Solving Olympics: An Inclusive Education Model for Learning Informatics. In *Informatics in Schools. Fundamentals of Computer Science and Software Engineering. Lecture Notes in Computer Science (ISSEP 2018)*, Sergei N. Pozdniakov and Valentina Dagienė (Eds.), Vol. 11169. Springer International Publishing, Cham, 319–335.

- **Lodi [2019]. Does Studying CS Automatically Foster a Growth Mindset?. In Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (Aberdeen, Scotland, UK) (ITiCSE '19). ACM, New York, NY, USA, 147–153.**

Reports

- **Nardelli, Forlizzi, Lodi, Lonati, Mirolo, Monga, Montresor, and Morpurgo [2017]. Proposal for a national Informatics curriculum in the Italian school. CINI.** *Proposal for a Grade1-Grade10 curriculum for teaching Informatics in school.*

Short Papers, Abstracts and Posters

- **Lodi [2017]. Growth Mindset in Computational Thinking Teaching and Teacher Training. In Proceedings of the 2017 ACM Conference on International Computing Education Research (Tacoma, Washington, USA) (ICER '17). ACM, New York, NY, USA, 281–282.**

- **Corradini, Lodi, and Nardelli [2018a]. Coding and Programming: What Do Italian Primary School Teachers Think? (Abstract Only). In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (Baltimore, Maryland, USA) (SIGCSE '18). Association for Computing Machinery, New York, NY, USA, 1074.** *Abstract and poster.*

- Lodi [2018b]. Pensiero Computazionale: dalle "scuole di samba della computazione" ai CoderDojo. *Mondo Digitale* 17, 77 (2018). *Abstract of the conference paper [Lodi, 2018c] republished on the Mondo Digitale journal.*

- Levrini, Barelli, Lodi, Ravaioli, Tasquier, Branchetti, Clementi, Fantini, and Filippi [2018]. The perspective of complexity to futurize STEM education: an interdisciplinary module on Artificial Intelligence. In *Abstracts of GIREP-MPTL 2018 conference* (Donostia-San Sebastian, Spain). 2 pages. *Abstract.*

- Branchetti, Levrini, Barelli, Lodi, Ravaioli, Rigotti, Satanassi, and Tasquier [2019]. STEM analysis of a module on Artificial Intelligence for high school students designed within the I SEE Erasmus+ Project. In *Proceedings of the Eleventh Congress of the European Society for Research in Mathematics Education (CERME11)*, Uffe Thomas Jankvist, Marja van den Heuvel-Panhuizen, and Michiel Veldhuis (Eds.), Vol. TWG26. Utrecht University, Freudenthal Group, Utrecht, Netherlands.

- Ravaioli, Barelli, Branchetti, Lodi, Satanassi, and Levrini [2019]. Epistemological Activators To Value S-T-E-M Concepts For Education. In *Proceedings the 13th Biennial Conference of the European Science Education Research Association* (Bologna, Italy) (*ESERA '19*). *Oral presentation.*

Books and Chapters

- Marchignoli and Lodi [2016]. EAS e pensiero computazionale. *ELS LA SCUOLA*, Brescia, Italy. *Theoretical book about situated learning and computational thinking. In Italian.*

- **Lodi, Davoli, Montanari, and Martini [2020]. Informatica senza e con computer nella Scuola Primaria. In Coding e oltre: informatica nella scuola, Enrico Nardelli (Ed.). Lisciani.** *Chapter in a collection of activities to teach Informatics. We describe the unplugged and plugged activities that we built and implemented in Bologna. In Italian. To appear.*

Prefaces

- Lodi [2016]. Pensare come un informatico non vuol dire pensare come un computer. In *Coding pensiero computazionale nella scuola primaria*. Marco Giordano and Caterina Moscetti (Auth.). ELI La Spiga. *Preface to a teachers guide structured on MIT creative learning approach, using Scratch. In Italian.*

Unpublished

- **Lodi and Martini [[n.d.]]. Computational Thinking, between Papert and Wing. Unpublished.**

Terminology

Specific terminology and relative abbreviations are always defined in the text (with appropriate bibliographic reference, when needed).

We report here the most important ones, as a handy cheat-sheet.

Computer Science / Informatics / Computing (CS) The terms are used as synonyms to refer to the *science* studying the algorithmic description and transformation of information. See the first section of Introduction for an in-depth discussion.

Computer Science (Informatics / Computing) Education (CSEd) The field studying the teaching and learning of CS. See Introduction.

Computer Science (Informatics / Computing) Education Research (CER) Research in CSEd field. See Introduction.

Computational Thinking (CT) The approach used to understand the world and solve problems through CS (inside the discipline itself, and in many other disciplines). Various interpretations and possible meanings are extensively discussed in Introduction, Chapter 2, Chapter 6 and Chapter 11.

K-12 education Roughly indicating pre-university education, from Kindergarten to Grade 12, i.e., the end of high school.

Pedagogical Content Knowledge (PCK) Teachers' interpretations and transformations of subject-matter knowledge in the context of facilitating student learning. See section Motivation in Introduction.

Content Knowledge (CK) Teachers' knowledge of representations of subject matter. Part of PCK. See section Motivation in Introduction.

Pedagogical Knowledge (CK) Teachers' general pedagogical knowledge or teaching strategies. Part of PCK. See section Motivation in Introduction.

Learning theory Philosophical, psychological or educational theory that describes *how people learn*. See Chapter 3.

Learning paradigms (educational paradigms / paradigms of education) Classifications of learning theories into schools based upon their most dominant traits. See Chapter 3.

Behaviourism Learning paradigm focused on shaping desired observable behavior and on the transmission of knowledge from teachers to students. See Chapter 3.

Cognitivism Learning paradigm shifting the focus from the behavior of students to their mental processes, and on the reorganization of information according to cognitive studies to foster learning. See Chapter 3.

Constructivism Learning paradigm focused on the idea that knowledge is built personally (cognitive constructivism) or socially (social constructivism) by learners rather than being transmitted to them. See Chapter 3.

Instructivism Term often used to generally indicate non-constructivist, traditional, transmissive paradigms and theories. See Chapter 3.

Constructionism Learning theory inside the paradigm of constructivism, focusing on the potential of computers and programming as tools to provide the (cognitive and affective) building materials useful to facilitate the construction of knowledge, even abstract one like the mathematical one. See Chapter 3.

Instructionism Used as the opposite of constructionism, i.e., using technology to improve teaching in a passive and transmissive (instructivist) way. See Chapter 3.

Mindset (implicit theories / self-theories / lay theories / naive theories) In this context, refers to personal ideas people hold about their intellectual abilities. See Chapter 5.

Fixed Mindset (entity theory of intelligence) Idea that own intelligence is a fixed trait one is born with (like eye color or height when adult), and one cannot do much to change it. See Chapter 5.

Growth Mindset (incremental theory of intelligence) (GM) Idea that own intelligence is a malleable trait that can be developed with study and deliberate effort (like muscles can be trained). See Chapter 5.

CS Mindset (CSM) Mindset with respect to Computer Science. See Section 5.2.

Transfer The effect of what you learned priorly in a new situation of learning or performance. See Chapter 4.

Higher-order thinking skills (HOTS) Include general skills like problem solving, critical thinking, creative thinking, and decision making. See Chapter 4.

Unplugged activities Activities without a computer, like physical games, used to teach CS and programming concepts.

Plugged activities The opposite of unplugged ones: activities that use computers (and programming) to teach CS concepts.

Programma il Futuro (PiF) project The italian localization of Code.org, with support materials. See Section 1.4.5, Chapter 10 and Chapter 11.

Introduction

Computer Science, Computing, Informatics, and Computational Thinking

Nowadays, digital technology pervades all aspects of human life and is based on an independent and recognized scientific discipline. This discipline is worldwide known as **computer science**, **computing** or **informatics**.

The term *informatics* is used, especially in Europe, as a synonym of *computer science* (and *computing*, common in the UK), rather than assuming a different nuance of meaning, as it sometimes happens nowadays in North America, for example. An interesting historical reconstruction by Bauer [2007] ascribes this different use of terminology to the fact that the name Informatics belongs to his company, and they denied to use it freely.

> The name Informatics has some historical significance. In 1973, my colleagues and I spent some time deciding on a name for our new company. We were attracted to the suffix "-atics," the Greek ending which suggests "the science of." [...] "Informatics" was our next thought, suggesting the "science of information handling." [...] Phillipe Dreyfus, a French system/software pioneer [...] invented the French version of the Informatics name: Informatique. In France, the name took on the meaning, generically, of "the modern science of electronic information processing." [...] The word Informatique was adopted and adapted in Europe in various forms: Informatik, Informatica, and soon. In the US [...] [t]he word belonged to Informatics legally [...] and through the years [...] we stopped many organizations from using the name. At one point, the Association for Computing Machinery asked us for permission to use the name. [...] I met Dreyfus [...] [and] we were amazed to learn that we had each developed the name in the same month and year, March 1962. [Bauer, 2007, p. 86]

Note, however, that the german *Informatik* appeared as early as 1957 in Karl Steinbuch's *Informatik: Automatische Informationsverarbeitung* (Informatics: Automatic Information Processing).

In the context of this dissertation, we assume the following definition.

Definition (Denning et al. [1989])**.** *The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, "What can be (efficiently) automated?"*

Note that in the above definition, "computing" was used as a shorthand for "computer science and engineering": "*We immediately extended our task to encompass both computer science and computer engineering, because we concluded that no fundamental difference exists between the two fields in the core material.*" [Denning et al., 1989, p. 10]. Nowadays

"computing" tend to be used as an umbrella term for a family of disciplines [Shackelford et al., 2006] (Computer Science, Computer Engineering, Information Systems, Information Technology, Software Engineering) or *fields that deal with computation* [Denning, 2013]: (*computer science, computational science, information science, computer engineering, software engineering)*.

Despite the fact we are fully aware of the different *nuances* of meaning, in this dissertation we will use interchangeably the terms *computer science*, *computing* and *informatics* (and the handy abbreviation CS) to refer to the *science* studying the algorithmic description and transformation of information. We will tend to use the term *informatics* in particular when talking about the Italian context and curriculum, since it resembles the Italian *informatica*, and is most common in Europe.

In the same way, since this dissertation is mainly about teaching and learning, we will use interchangeably the expressions *Computer Science Education* (CSEd), *Computing Education*, *Informatics education*. When talking about *Computing Education Research*, we will also use the acronym CER.

The approach used to understand the world and solve problems through CS (inside the discipline itself, and in many other disciplines) is nowadays often referred to as *computational thinking* (CT) [Wing, 2006]: for the moment, it can be informally defined as *thinking like a computer scientist to solve problems*. We will discuss in much depth the term, its origins, and its many possible definitions in Chapters 2 and 6.

Motivation

We live in a digital society, where computing devices are ubiquitous, and where all aspects of our lives - from work, to socialization, to entertainment - are extensively influenced by artifacts (both hardware and software) built thanks to the results of Computer Science.

It is no surprise that the introduction of scientific and cultural aspects of Computer Science (CS) in K-12 education is currently of great interest for stakeholders, policymakers, educators, and, of course, CS Education researchers.

Some governments are therefore taking actions, by introducing CS aspects (sometimes calling them CT or using the buzzword "coding") in K-12 curricula and teaching guidelines. At the same time, volunteers and private associations and companies, are making an effort to spread out basic CS principles (often framed as the teaching of practical programming skills).

The interest of this author grew in these two contexts: from one side becoming (only for one year) a CS High School teacher, and from the other by volunteering in CoderDojo Bologna.

As governments quickly found out [see, e.g., The Royal Society, 2017], one of the most important (and too often least considered) aspects of putting in place a stable system of CS in K-12 is focusing on teachers.

It is clear to researchers that, when a new discipline is introduced in the curriculum, training (for pre-service teachers) and professional development (PD) (for in-service teachers) are crucial aspects. In particular, according to the model presented by Bender et al. [2015], relevant areas for training and PD are:

- Pedagogical content knowledge
- Teachers' beliefs
- Motivational orientations

Even though this thesis is not focused only on teachers, this framing is very useful to understand what areas are worth researching to foster the introduction of CS in K-12 education.

Pedagogical content knowledge. Pedagogical content knowledge (PCK) was defined by Shulman [1987] as teachers' interpretations and transformations of subject-matter knowledge in the context of facilitating student learning. He proposed several key elements of pedagogical content knowledge:

- knowledge of representations of subject matter (CK, content knowledge);
- understanding of students' conceptions of the subject and the learning and teaching implications that were associated with the specific subject matter;
- general pedagogical knowledge (PK) or teaching strategies;
- curriculum knowledge;
- knowledge of educational contexts;
- knowledge of the purposes of education.

As most of the teachers are not specifically trained to teach CS, and probably had not received any formal instruction in computer science (as opposed to other subjects, like Math, that has been studied from primary school by all teachers), training initiatives should consider the necessity to teach elements of computer science (CK), along with specific strategies to teach those concepts. This is even harder because there is no long tradition in teaching CS, and so there is an ongoing debate and research to determine what concepts and competencies should be taught in K-12 education, and at what levels should each concept be introduced.

Pedagogical knowledge (PK), however, must not be forgotten: as we will review, there is much debate between pedagogical paradigms, for example, the classic battle between traditional instructivist approaches, focused on how to transmit knowledge, and more active and constructivist approaches, focused on fostering students constructing knowledge by themselves.

Teachers' beliefs. Teachers' conceptions about a discipline are fundamental elements for its teaching. In our society there are many misconceptions on what *computer science* is (e.g., confusing it with the use of digital technologies), and nowadays the spread of new terms like "computational thinking" and "coding" can lead to misconceptions among instructors (e.g., considering "computational thinking" as a separate discipline from CS, or "coding" as more general and didactic than "programming").

Apart from erroneous ideas about what CS is, other misconceptions are spreading, regarding transversal competences (like general problem solving, logical reasoning, creativity, collaboration, perseverance) the study of CS can foster. There is a long history in claims about what skills can be developed by teaching programming, dating back in the 80s, and focusing on the educational language LOGO [Papert, 1980]. Research in education, however, tells us transfer is difficult and unlikely to happen automatically, especially between knowledge domains far from one another [Ambrose et al., 2010].

Motivational orientations. Personal ideas (among present and future teachers, but also among students) about the ability of someone to learn computer science are fundamental for an effective introduction of CS as a discipline in the school system.

To non computer scientists, learning to program may appear as a too challenging goal, achievable only from those having the so-called "geek gene" [Ahadi and Lister, 2013; Patitsas et al., 2016]. Moreover, stereotypes lead some people to identify computer scientists with singularly focused, asocial, competitive, male figures [Lewis et al., 2016].

Students and teachers have different personal ideas ("implicit theories") about their intellectual abilities [Dweck, 2017b]. Some believe that their intelligence is a fixed trait (like eye color or height when adult), and they cannot do much to change it: they have *fixed mindset*. Some others believe instead that intelligence can be developed with study and effort (like muscles can be trained): they have a *growth mindset*.

A growth mindset can be fundamental to face the challenges of learning to program, and this is particularly true for primary school teachers, mostly female (and so, subject to gender stereotypes) and not trained in CS and its teaching. It is known one can have a different mindset for different areas: so it is important to measure and address a specific "CS mindset."

Research presented in this thesis covers aspects of each of these areas.

Part I is mainly dedicated to providing the reader a brief but precise review of the main concepts (both from computer science and from psychological and educational literature) involved in this dissertation, to make it sufficiently self-contained. The reviews will be descriptive rather than critical. However, we will provide the first contribution: a classification of some important elements of CT that, we argue, is helpful in shedding light on misinterpretations of the concept.

After that, the three main parts of this document will cover the aspects researched in the aforementioned areas.

Part II covers historical, epistemological, and pedagogical aspects of the nature of Computer Science as a discipline, and its introduction in K-12 education.

Part III covers teacher's sentiment about a nationwide project to introduce CT, and conceptions and misconception new expressions ("computational thinking," "coding") used in the educational context.

Part IV investigates the relationship between implicit theories of intelligence (mindset) and teaching and learning CS.

Finally, Part V contains concluding remarks, appendixes, and bibliography.

Research Goals

This research was guided by an important *meta research question*.

MRQ *What are the positive and negative effects of historical, epistemological, pedagogical, cognitive, and affective aspects of CT on the introduction of CS in K-12 education?*

More specifically, we were guided by some fundamental questions, focusing on three crucial areas.

Historical, philosophical, and pedagogical aspects. What are the core epistemological aspects of CS worth being included in a K-12 CS curriculum? What are the historical and pedagogical contexts in which the idea of CT emerged, and what can we learn today from them? What is the relationship between learning theories (e.g., constructivism) and teaching approaches (e.g., educational programming languages, unplugged activities)?

Cognitive aspects. The introduction of a new discipline can come with a lot of stereotypes and misconceptions, especially among generalist teachers not trained to teach the discipline. What is the sentiment of those teachers? What conceptions and misconceptions do they hold about concepts and terms like CT and "coding," intensively used by media, trainers, and ministerial documents?

Affective and motivational aspects. What is the impact of personal beliefs about intelligence and about own CS ability in teaching and learning CS? What is the role of teaching approaches (e.g., constructionism)?

Investigations on these questions have been conducted both through historical and philosophical argumentation, and through quantitative and qualitative research on collected data (on large nationwide samples and on small significant samples).

Moreover, many of these questions have been looked through the lens of the (often exaggerated and too optimistic) claims about the (automatic) *transfer of competences* (e.g., from studying CS to learning general problem-solving skills, or from learning CS to applying perseverance in all areas of life).

Contributions

It should be clear that it is not possible to give definitive answers to such big questions, especially in the narrow scope of a Ph.D. thesis.

However, the results of our work can provide support for valuable claims regarding the introduction of K-12 in CS education, and bring contributions to CSEd research.

The claims are briefly stated here, and will be extensively analyzed in the Conclusions at the end of this document (Chapter 14).

- CS should be introduced in K-12 education as a tool to understand and act in our digital world, and to use the power of computation as a tool for thinking and meaningful learning. CT is the (important) conceptual sediment of that learning. We designed a curriculum proposal in this direction.

- The use of expressions like "computational thinking" (which is useful to make clear that we are not talking about digital literacy) and "coding" can cause misconceptions and partial understandings between non-specialist teachers, often focusing on unverified claims about transfer to general thinking skills, hence in need of specific training, both on pedagogical aspects and, first of all, on CS disciplinary content.

- Programming practices, teaching tools like visual programming environments, and unplugged activities have some intrinsic constructivist and constructionist characteristics (e.g., visualization, sharing and remixing, creative construction of interactives, concept reconstructions, personalization, use of the body) that can be used to facilitate CS learning, as shown with activities examples we designed.

- Like other general skills, growth mindset is not automatically fostered by learning CS. Worse, not studying CS can foster fixed beliefs about CS ability. A growth mindset could, however, be fostered by creative computing activities, leveraging on the constructivist aspects of CS.

Thesis Overview

In the next subsections, which can be safely skipped if reading the entire material, we will present in more detail each chapter. Beware of spoilers!

Context (Part I)

In the first part of this work, the context is depicted.

We will start, in Chapter 1, from a description of the social and cultural context in which the introduction of Computer Science in K-12 is gaining importance. We will describe initiatives from governments that are introducing CS concepts in curricula (e.g., US, UK, France) and non-profit organizations (e.g., Code.org, CoderDojo), aiming to widespread basic CS skills (often through a playful introduction to programming concepts).

We will then focus on the Italian situation, describing - especially for the unfamiliar reader - the Italian school system, with particular focus on curricula and recent ministerial documents related to CS and ITC, and on the pre-service primary teacher training. Finally, we describe the project "Programma il Futuro," the Italian localization of Code.org.

In Chapter 2, we will explore the expressions "computational thinking" and "coding," hot keywords massively used when talking about CS in K-12. We will focus on the current meanings of the expression CT[1], and discuss the many definitions that have been proposed

[1] leaving historical research of its origins for Chapter 6

in recent years. In particular, we summarised the common aspects found in the definitions, distinguishing four areas [Corradini, Lodi, and Nardelli, 2017b]:

- **Mental processes**: mental strategies useful to solve problems (*Algorithmic thinking; Logical thinking; Problem Decomposition; Abstraction; Pattern recognition; Generalization*).
- **Methods**: operational approaches widely used by computer scientists (*Automation; Data Collection, Analysis and Representation; Parallelization; Simulation; Evaluation; Programming*).
- **Practices**: typical practices used in the implementation of computing machinery based solutions (*Experimenting, iterating, tinkering; Test and debug*).
- **Transversal skills**: general ways of seeing and operating in the world; useful life skills that could be useful for / enhanced by thinking like a computer scientist (*Create; Communicate and collaborate; Reflect, learn, meta-reflect; Be tolerant for ambiguity; Be persistent when dealing with complex problems*).

This classification is important because some of the aspects tend to be overemphasized (e.g., general problem solving mental processes and transversal skills) and other underemphasized (e.g., CS specific methods and practices), as will appear in different parts of this dissertation.

Finally, we will discuss different views on the current meaning of the term "coding": from one phase of software development to a synonym of programming, up to "playful programming for beginners."

In Chapter 3, we will set some pedagogical background needed to understand this research: after a brief introduction about learning paradigms, we will describe more deeply those relevant for our work. We will start from the classical *behaviourism*, main theory behind traditional transmissive (*instructivist*) approaches to teaching; we will then discuss *cognitivism*, dismissing the behaviorist assumption that cognitive processes cannot be studied; we will then move to *constructivism*, a set of psychological and learning theories sharing the idea that knowledge is actively constructed or reconstructed by learners (or groups) rather than being transmitted to them.

After these classical paradigms, we will focus deeply on *constructionism*, a constructivist theory that is tightly bonded with CT and learning to program: it adds that learning is especially effective when the learner is involved in the active construction of objects - both cognitively and affectively meaningful to her. To construct these objects, she needs *building materials* (concrete or abstract): computers and programming languages are formidable tools to provide these materials.

We will finally present a learning model, *creative learning*, developed in the context of constructionism, that inspires the widespread educational blocks-based programming language Scratch.

Chapter 4 is dedicated to a review on a single crucial concept in Education (and one of the threads of this dissertation): transfer of learning. Informally, it can be described as *the*

effect of what you learned priorly in a new situation of learning or performance. There is a long tradition of claims that the so-called *higher order thinking skills* (HOTS), like general problem solving, logical thinking, problem decomposition, can be taught through disciplines like Math, Latin or Greek and then they automatically transfer to other disciplines or even to every aspect of human life.

In these years, we see a similar trend in our field: mostly unverified claims state that studying CS can automatically improve students' results in other disciplines, or teach them domain-general problem-solving skills (e.g., abstraction, pattern recognition, problem decomposition, logical reasoning) or influence cognitive and affective aspects (e.g., motivation, self-efficacy, resilience, creativity, mindset).

These claims are not new to computer science education research: debates dates back to as early as the 1980s, in the context of the educational language LOGO, developed by Seymour Papert. Some research and reviews were conducted in the early 90s, leading to negative or only slightly positive conclusions.

More encouraging results were, however, obtained when the interventions were explicitly designed to transfer specific competencies. In fact, educational research warns against automatic transfer and agrees transfer is difficult, especially between far domains and especially if activities are not explicitly designed for it.

Transfer is likely to happen more easily between near domains and applications. Less likely is that transfer will automatically happen between CS and HOTS, whose teachability is highly debated, or, even worse, to broader aspects like growth mindset.

Moreover, different teaching approaches and tools can influence transfer. From one side, a more abstract and top-down instruction seems to foster transfer, since it teaches connections and applicability of the same knowledge in different situations, but, on the other side, a more hands-on bottom-up approach (e.g., a constructivist approach) seems to help creating a more profound and more retainable knowledge.

Finally, successful examples of using CS (and in particular programming) mainly as a tool to foster the learning of other subjects are available: however, the amount of computer science learned by students is confined to what they strictly need to use in the main subject that they are learning.

In Chapter 5, we will focus on mindset theory, concerning the effects of personal ideas that people hold about their own intellectual abilities. Having a growth mindset (believing that intelligence can be trained, like muscles) is far more desirable than having a fixed mindset (believing intelligence is an immutable trait like eye color). Students and teachers with a growth mindset get better results, cope better with difficulties, are less subject to stereotypes. Growth Mindset can be taught by specific interventions.

This cognitive theory has recently gained much attention among educators around the world. For what regards CS, the (unproven) idea that studying CS can foster a growth mindset emerged.

We will review significant findings related to the theory and its implementation in schools, also acknowledging doubts and critics that some researchers raised about it, and discuss possible explanations of the current research status.

For what concerns CS, only a bunch of studies have been conducted, leading to mixed

and inconclusive results. Moreover, it is accepted that one can have different mindsets with respect to different disciplines, so we considered in particular CS mindset, the mindset with respect to computer science. Research in other disciplines seems to indicate that constructivist approaches can be beneficial for fostering a growth mindset, and this is confirmed by our preliminary study.

History, Epistemology and Pedagogy (Part II)

Chapter 6 proposes a historical and philosophical analysis [Lodi, 2018c; Lodi and Martini, [n.d.]] of the original context in which the expression "computational thinking" firstly appeared: Seymour Papert's attempts to use programming language LOGO as a tool to constructively learn math, geometry, physics and potentially any other subject.

After a brief historical account on the *idea* of computational thinking, present in the discipline from its beginning at the end of the fifties, we will analyze all the occurrences of the expression "computational thinking" in Papert's writings. This will give us a better understanding of constructionism, his educational theory.

This analysis shows that one of the main points in what we informally call *Papert's CT* is that computers and programming languages are formidable meta-tools for building computational objects in an environment rich of computational principles and meaningful (both cognitively and affectively) for the learner and the community. In this context, CT is seen as the privileged way to learn all the other subjects by creating significative artifacts through programming.

However, history showed this has been oversimplified and misunderstood by thinking (and sometimes successfully rebutting) that Papert's claim was about "programming automatically enhancing general problem solving and thinking skill." As we already said, this idea has a long and persistent history.

We then analyze differences with what can be called *Wing's CT*: originally informally defined by Jeannette Wing as "thinking like a computer scientist" and then as "the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent." However, we argue [Lodi, Martini, and Nardelli, 2017] that the reason CT must become common knowledge is that it helps to understand and actively participate in our digital society, more than - as was initially put by Wing - it helps to apply CS problem solving to everyday aspects of human life (e.g., to better arrange your backpack).

We propose to focus instead on the dual nature of CS, also emerged from Papert's and Wing's views: from one side, CS is an independent scientific discipline with his own core scientific concepts, that is fundamental to learn - as we do for other subjects - to understand the world we currently live in; from the other side, CS can be a powerful interdisciplinary and transversal tool, with its unique ability to automate abstractions, to simulate worlds and to allow everyone to express their creativity as constructors of digital artifacts.

These ideas form the basis for a proposal of a curriculum, detailed in Chapter 7 (and fully reported in Appendix B) for CS in Italian compulsory education: primary (grades 1 to 5), lower secondary (grades 6 to 8) and first years of upper secondary (grades 9 and 10).

The proposal has been inspired by the English curriculum, but has been written to match the structure and the style that the current Italian guidelines adopt for the other subjects. It is articulated in five areas: Algorithms, Programming, Data and Information, Digital Creativity and Digital Awareness, clearly putting the focus on scientific aspects of CS, without forgetting its power as a creative tool [Nardelli, Forlizzi, Lodi, Lonati, Mirolo, Monga, Montresor, and Morpurgo, 2017; Forlizzi, Lodi, Lonati, Mirolo, Monga, Montresor, Morpurgo, and Nardelli, 2018].

In Chapter 8, we will review some relevant literature related to learning CS and, more specifically, programming in a constructivist and constructionist light. First of all, we present some cognitive aspects, for example, the ideas of notional machine and its role in understanding, misunderstanding, and difficulties in learning to program. We will then review programming languages for learning to program, with particular focus on educational characteristics of block-based languages. We will also explore similarities (e.g., iterative software development and constructive, bottom-up, building of knowledge) and differences (e.g., well defined abstract machine executing code vs. rebuttal of the existence of objective reality) between some aspects of CS and of constructivist theories [Monga, Lodi, Malchiodi, Morpurgo, and Spieler, 2018; Lodi, Malchiodi, Monga, Morpurgo, and Spieler, 2019].

We will then present the widespread but debated pedagogical approach of "unplugged activities" [Bell and Lodi, 2019a]: activities without a computer, like physical games, used to teach CS and programming concepts. We will discuss the advantages and limitations of this kind of activity.

We will focus in particular to the most famous set of activities: CS Unplugged [Bell and Lodi, 2019a;b]. It was developed many decades ago in New Zealand as outreach material, but in recent years is being more and more used in schools. The similarities (e.g., kinesthetic activities) and differences (e.g., structured vs. creative activities) between Unplugged and constructivist/constructionist activities are highlighted.

Also in this context, the transfer problem is fundamental: do the concepts (e.g., algorithms) learned through CS unplugged automatically transfer to "plugged" contexts (e.g., an implementation in a programming language)? Research is consistent in showing that these activities are helpful, and can lead to equal or better learning, only if integrated with "plug it in" activities, where the concepts are explicitly linked to the ones learned in unplugged activities.

In Chapter 9, we sum up the proposals of "unplugged+plugged" activities by describing teaching materials and lesson plans that we produced and implemented in a primary school [Lodi, Davoli, Montanari, and Martini, 2020].

The unplugged activities are structured as an incremental discovery, scaffolded by the instructors, of the fundamental concepts of structured programming (e.g., sequence, conditionals, loops, variables) but also complexity in terms of computational steps and generalization of algorithms, through playing cards and moving characters on a grid.

The plugged activities follow the creative learning approach, using Scratch as the main tool, both for free creative expression and for learning other disciplines (e.g., drawing regular polygons). Moreover, awareness of the rules of a digital community is addressed.

All the activities are explicitly linked to the competence and knowledge goals presented in

our curriculum proposal (Chapter 7).

Teachers' Conceptions (Part III)

In this part, we switch from a philosophical, epistemological, and pedagogical discussion to a more narrow empirical research on large scale samples involving primary teachers enrolled in the national project Programma il Futuro.

In chapter 10, we analyzed [Corradini, Lodi, and Nardelli, 2017a] primary school teachers' ($N \approx 1300$) sentiment after the first two years of the project. Responses were largely positive.

It is worth noticing that some of the most positive aspects of the project reported by teachers were mainly related to the transversal skills (rather than to CS core concepts) they feel these activities can improve in their students, for example: promotion of awareness and comprehension of problem solving, logical thinking, creativity, attention, planning ability, motivation for learning, students interest, cooperation.

Interestingly, we found that teachers perceived that the interest in their students with respect to gender is equally distributed in primary school, but tend to increase for boys and decrease for girls from secondary school. This reinforces the idea that it is essential to introduce CS from primary school, when gender stereotypes are not yet present.

Teachers giving negative answers, even if they were a minority, raised questions about the lack of creativity in the activities proposed by Code.org (that are, in fact, more oriented to computational problem solving than on the creation of artifacts) and, above all, asked for more professional development.

In chapter 11 we investigated teachers' ($N \approx 1000$) conceptions and misconceptions on the two widespread expressions "computational thinking" and "coding".

First of all [Corradini, Lodi, and Nardelli, 2017b], we asked teachers what is computational thinking, what is the relationship between CS and the use of technology and finally how much they felt prepared to teach CS basics.

The study showed only 1% of teachers gave a "good" definition, and 10% of them gave an acceptable one. The others gave partial definitions, recognizing some important aspects (like "problem solving" or "transversal competence"), but without referring to algorithms, programs or executor, and so stepping away from a "CS specific" characterization of the concept, reinforcing the worries about the spread of misinterpretations about CT.

On the other hand, it is encouraging the fact that 80% of them are aware of the distinction between "computational thinking" and "being able to use technology."

Not surprisingly, 2 out of 3 teachers felt not prepared or scarcely prepared to teach CT, and ask for professional development as the first element to improve their preparation.

In the same study, we also investigated [Corradini, Lodi, and Nardelli, 2018a;b] the perception of the difference between the term *coding* (widely used in media, ministerial acts and by some teachers and instructors) and the term *programming*.

When asked about their idea of "what coding is," only 40% of the answers related coding and programming, but another 19% used terms linked to the automatic elaboration of information to describe what coding is. The remaining part of the sample did not provide explicit or implicit links between coding and programming.

Worrying are the misconceptions we found in those who think coding is different from programming: some think coding is just "toy programming," while others state it is "more abstract and general" than programming, linking it again to the development of transversal problem solving skills rather than to CS core concepts.

This investigation shows that untrained teachers hold misconceptions regarding CS and its related terms. Given the general public and media attention on CT and "coding" in schools, currently taught by existing teachers - mostly not appropriately trained, professional development actions focusing on CS scientific principles and methods are, therefore, a top priority for the effectiveness of CS education in schools.

Implicit Theories (Part IV)

In this part, we will focus on the relationship between growth mindset (GM) and CS education. In fact, as already stated, untested claims about CT fostering transversal skills like resilience and GM are emerging.

However, educational research shows that transfer of competences is hard. Very little research has been conducted on the relationship between GM and CS learning, with conflicting results.

To investigate this connection, in Chapter 12 we report on an experiment [Lodi, 2019] where we measured some indicators (e.g., mindset, computer science mindset - that we defined here as *the mindset with respect to CS*) at the beginning and the end of a high school year in five different classes (N = 66): three CS oriented, one Chemistry oriented, and one Transportation&Logistics oriented. In one of the CS oriented classes, we did a very brief GM intervention.

At the end of the school year, none of the classes showed a statistically significant change in their mindset, reinforcing the idea that a transversal skill like mindset is hard to transfer automatically.

Interestingly, non-CS oriented classes showed a significant decrease in their computer science growth mindset. This is not desirable: if we think CS has a universal social value, reinforcing stereotypes about a "geek gene" will be harmful and lead to problems similar to those that many students are experiencing in Math, as international tests reveal. This adds to the evidence on the importance of introducing CS principles for all K-12 students.

In the intervention class, students suggested - to stimulate a GM - the need for activities that are more creative, engaging, and related to the real world and their interests, which is in line with research on GM in other fields and shows in the constructivist/constructionist/creative approach a way to pursue to enhance CS education.

The importance of a creative approach seems to be confirmed, as is described in Chapter 13, by a preliminary study (N = 43) we conducted in female pre-service primary teachers (master students) before and after a "creative computing with Scratch" course. We found a statistically significant, even though little, increase in their growth mindset [Lodi, 2017; 2018a].

The course content and methodologies are described and discussed, in the light of fostering a growth mindset in pre-service teachers, for example, by leveraging on the creative and iterative nature of CS.

Conclusions (Part V)

In Chapter 14 we review our results - acknowledging their limitations, and propose further direction this research opens, for example: the design and evaluation of constructivist activities keeping together the two natures of CT; the design of teacher training/PD focusing also on CK; research on transfer in modern contexts; new methodologies to evaluate CS mindset.

In Appendixes, we present additional material: other classifications of CT concepts (A), our curriculum proposal (B) and the instruments used in our mindset experiments (C and D).

Part I

Literature Review

Chapter 1

CS in K-12 Education[1]

In this chapter, we argue for the importance of studying CS in K-12 schools because of the ubiquitous role it has in our society.

We discuss how many countries are introducing it in school curricula and how private initiatives support the cause as well.

Then, we focus on the Italian system, curricular documents, and the situation of teacher training. We finally present the "Programma il Futuro" project, Italian localization of Code.org, that is the background of some studies we conducted.

1.1 The Importance of CS in Society and in Schools

Computer Science is no longer a subject area cultivated only by professionals, but is relevant to every citizen. CS is having, and will continue to have, a growing impact on the development of production, economy, health, science, culture, entertainment, communication, and society in general.

During its evolution, computer science developed its own big ideas. For example, following Bell et al. [2018]: "information is represented in digital form," "programs express algorithms and data in a form that can be implemented on a computer," "digital systems create virtual representations of natural and artificial phenomena," "digital systems communicate with each other using protocols," and so on.

Every science has its particular point of view on the world, its "conceptual paradigm," according to which it describes and explains the phenomena. So - to give some examples - quantity and their relationships are essential concepts for a mathematician, forces and masses for a physicist, organism, metabolism, and reproduction for a biologist, molecules and their reactions for a chemist, and so on. Each of these paradigms can be used to describe the same reality, and - depending on the cases and contexts - one (or the combination of some) of them may be the most useful for understanding and explaining the observed phenomena.

[1] Minor parts of this chapter were already published in introductory sections of Lodi, Martini, and Nardelli [2017], Corradini, Lodi, and Nardelli [2017a; 2018b], Forlizzi, Lodi, Lonati, Mirolo, Monga, Montresor, Morpurgo, and Nardelli [2018], Lodi [2019].

Concepts such as algorithm, language, automaton are part of the conceptual paradigm of CS and can be successfully used to provide descriptions complementary to those provided by other sciences. Denning and Rosenbloom [2009] have coined the expression "the fourth great domain", putting computing on par with physical, life, and social sciences as a way to grasp reality [Denning, 2009].

This different perspective that CS has on the world has also enormous practical consequences: to pick one of the many possible examples, studying computational complexity allowed the development of cryptography, giving us the opportunity for secure credit card transactions.

Knowing the fundamental concepts of the science at the base of these marvels of digital technology allows us - again, just to make examples - to discuss with greater awareness of the advantages and threatens of electronic voting and in general of online data security, to better understand the effects of (real or fake) news dissemination on social networks, understanding the effects of algorithms governing every aspect of our lives - from dating suggestions to the automated stock exchange, to artificial intelligence applications like self-driving cars, and so on. Furthermore, considering that in the current digital society the presence of IT devices is ubiquitous, it is necessary for any citizen who wants to be at ease in the society itself to understand the scientific principles underlying their operation.

In this, computer science has not only a descriptive and a hermeneutic role, but it also allows one to act on reality. Whether it is couriers that use algorithms to manage better the logistics of constantly growing deliveries, or historians who use natural language processing techniques to make new hypotheses on symbols that have not yet been deciphered, CS is a necessary tool for transforming the world according to our own purposes, in every work activity and every scientific discipline and humanities.

In order to cope with the ubiquity of information technology, all citizens must acquire the conceptual tools necessary to understand the science underlying the digital world in which they are immersed and on which the quality of their life will depend. Although we are experiencing a rapid evolution of digital devices and of their applications, CS scientific foundations are firm and rest on a homogeneous range of concepts, methodologies, and skills.

In the last decade, the awareness that these principles are valuable for every human being has increased, and the introduction of CS as a standard school subject is more and more discussed and advocated [see, e.g., Hromkovič, 2006; 2016]. At school, one learns Physics, Biology, History, Literature not (necessarily) to become a scientist, a writer, and so on, but to understand the world one lives in. As said, finding out what is behind technologies allows students to become informed citizens, and to better debate and decide on crucial issues like genetics, privacy, e-vote, and so on. Moreover, a lot of the so-called "digital jobs" are unfilled because of the lack of a prepared workforce. To cover this vacancy, a broad education in computing is mandatory. Less general but still very relevant, to increase the number of students (and in particular underrepresented categories such as women) who choose to graduate in Computing disciplines, an early exposition during K-12 to the basis of CS is required so as they can fully understand and - if the case - appreciate it.

Regardless of the approach or tools proposed to teach CS in K-12, which are highly debated, "*there is a convergence towards computational thinking as a core idea of the K-12*

curricula" and "*programming in one form or another, seems to be absolutely necessary for a future oriented*" CS education [Hubwieser et al., 2014]. As far as the learning of programming is concerned, we should also note that its scope is now broader than it used to be. Indeed, researchers argued that practicing programming can be seen not only a technical skill, but also as a means of self-expression and social participation [Kafai and Burke, 2013; Schulte, 2013], as a component of a new form of literacy [Burke, 2012; Vee, 2013], as a way to widen experience and experiment with personal ideas [Boyatt et al., 2014], and maybe also as an instrument to foster children's metacognition [Resnick et al., 2009].

1.2 CS in School Curricula

CS at school is often misrepresented as mere use of digital technologies, but this is, of course, a distorted view. As seen, its real educational value, both as an independent *scientific* discipline and as a cross-disciplinary field, lies on the fact that it offers new and meaningful ways to interpret the world around us and to approach problems. The terms *computational thinking* (CT) and *coding* (that we will deeply discuss in Chapter 2) are often used nowadays from governments, stakeholders and educators. However, as we will extensively argue throughout this thesis, we think students need an adequate CS education actually to acquire CT and fully participate in the digital society.

The role of computing in school curricula is currently a topical issue of education policies all over the world. CS has recently been introduced - or is on the verge of being introduced - in the official curricula for compulsory education of several countries.

In the US, for instance, the Computer Science Teachers Association (CSTA), in cooperation with the ACM, has proposed comprehensive standards [CSTA, 2017] and frameworks (with the further collaboration of CIC, NMSI and Code.org [K-12 CS Framework, 2016]) for K-12 education. The legislative act "*Every Student Succeeds Act*" (ESSA), approved by the Congress in 2015 with bipartisan support, introduced information technology among the "*well rounded educational subjects*" that must be taught in school. In January 2016, the then President Obama launched the initiative "*Computer Science For All*"[2] whose goal is "*to put all American students, from kindergarten to high school, in a position to learn computer science*".

In the UK, following the exhortations from the The Royal Society [2012] Report "Shut Down or Restart" (denouncing the failure of the previous curriculum, based on the *use* of ICT), Computing is a mandatory subject for all instruction levels starting from s.y. 2014-15 - see in particular England's "Computing at School" curriculum [Computing at School, 2012].

In France, the report by the Committee on Science Education [2013] advocated the introduction of computer science at school. In 2015, the France government did reform the curriculum of compulsory school introducing programming in cycle 3, and programming & algorithmics in cycle 4[3].

Similar debates and changes in curriculum are underway or have taken place in several other countries all around the world (just to mention some of them: most of the EU countries,

[2] www.csforall.org

[3] https://www.education.gouv.fr/pid285/bulletin_officiel.html?pid_bo=33400

Israel, South Korea, Japan, Australia, New Zealand), although there is not yet full consensus as to what should be taught in K-12.

The associations Informatics Europe and ACM Europe have jointly put forth to the European Commission the "Informatics for All" proposal [Caspersen et al., 2018], whose aim is to establish Informatics as an essential discipline for students in Europe at all levels throughout the educational system. The proposal adopts a two-tier strategy. A first-tier takes the form of informatics as a specialization, i.e., a fundamental and independent school subject. The second tier would be the integration of informatics with other school subjects.

A broad picture of the state of CS education worldwide can be found, for example, in Hubwieser et al. [2011], McCartney and Tenenberg [2014], Barendsen et al. [2015], whilst the situation is constantly changing.

1.3 Other Initiatives

In recent years, no-profit organizations, volunteer movements, and also private initiatives aiming at spreading CS to young people flourished. Most of them are of high quality and have helped to gain institutions and media attention on the topic. We believe, however, that - in the long run - they cannot afford to provide CS education to all: K-12 public education system has to take in charge of this ambitious target.

Some of these organizations provide after school programs or clubs to teach programming and creative computing (e.g., CoderDojo[4,5], CodeClubs[6]), sometimes with specific targets (e.g., Girls who code[7], Black Girls Code[8]).

Other are communities of teachers or educators (e.g., the British CAS - Computing at Schools[9], or the ScratchEd[10] community from Harvard).

There are also initiatives in the form of challenges proposing problem-solving activities related to CT and informatics: for example, the Bebras[11] challenge [Dagienė and Sentance, 2016], born in Lithuania and now held in more than 60 countries around the world [Dagienė, 2018], or the Problem Solving Olympics[12], supported by the Italian Ministry of Education [Borchia et al., 2018].

Finally, there are political initiatives to raise awareness and encourage students, teachers, educators, and parents on the importance of learning CS/programming (e.g., the "Hour of Code"[13] from Code.org - see next section, or the "EU Code Week"[14]).

[4] https://coderdojo.com/

[5] The author of this thesis is a proud member of CoderDojo Bologna, to which he wishes he could dedicate more time.

[6] https://codeclub.org/

[7] https://girlswhocode.com/

[8] http://www.blackgirlscode.com/

[9] https://www.computingatschool.org.uk/

[10] https://scratched.gse.harvard.edu/

[11] https://www.bebras.org/

[12] https://www.olimpiadiproblemsolving.it/

[13] https://hourofcode.com/

[14] https://codeweek.eu/

1.3.1 Code.org

Until 2013, very few USA states had computer science in school curricula, despite being probably the most advanced country in IT technology. To counter this situation, the no-profit organization Code.org[15] launched in the same year the "Hour of Code" project, with the initial goal of having each student in the world doing at least one hour of programming and, in perspective, the final goal of having for each student a proper education in computer science.

Code.org developed teaching material made up of online interactive web programming tutorials (see 8.3.6.2), featuring famous video games and cartoon characters, highly attractive for students. Web tutorial are interspersed with *unplugged* activities (see 8.4) that teach or reinforce important informatics concepts. Printable material and detailed lesson plans are provided. Teaching materials and curriculum progression have been defined keeping in mind the K-12 CS Framework [2016].

Code.org had, in its first school-year alone, more than 40 million students doing their first hour of coding all around the world, and each year the participation dramatically increased.

1.4 The Situation in Italy

1.4.1 The Italian School System

Starting from 2007, the Italian school system has undergone a broad reform process, aimed at renewing both the educational approach and the curricular organization. Compulsory education spans now over ten years, usually corresponding to the age range 6–16, and is subdivided into three main stages: primary school (grades 1–5), lower secondary school (grades 6–8) and early upper secondary school (grades 9–10).

1.4.1.1 Primary and Lower Secondary

In 2012 the Italian Ministry of Education, University and Research (MIUR) issued the curricular recommendations for the primary and lower secondary levels, that are common to all schools. As opposed to the previous instructional programs, where the content of each subject area was mainly arranged in temporal sequence, the new framework aligns with the recent European trends in pre-tertiary education, focusing on skills and competencies to be acquired in broad areas[16].

1.4.1.2 Upper Secondary

The upper secondary level, on the other hand, is characterized by a variety of strands, whose curricula are substantially differentiated from the outset, as appears from the list of specific documents.

[15] https://code.org/

[16] In particular, the Italian Ministry has adopted the "Recommendation of the European Parliament and of the Council" of 18 December 2006 on key competences for lifelong learning (2006/962/EC).

Upper secondary school lasts five years (usually students start when they are 14 y.o. and finish when they are 19 y.o.), but only the first two years are compulsory. Students can choose between **lyceums**, which give a theoretical basis in classical, scientific or artistic areas and naturally lead to university studies; **technical institutes**, which give both theoretical basis and a high qualified specialization in a specific area (divided in: *economic institutes*, preparing for economics, management, and business-oriented computer programming, and *technological institutes*, preparing for areas such as mechanics, electronics, computer science, chemistry), leading to a job or to university studies; or **professional institutes**, preparing students with practical skills to enter the job market immediately.

After choosing the kind of school, students select a specific track offered by the school (e.g., "IT and telecommunications," "fashion"), whose curriculum is decided at a national level. Almost every curriculum includes Italian, English, History, Maths, Natural Sciences, Sports, and the subjects of the chosen track.

See Bellettini et al. [2014] for a more comprehensive summary of the secondary school system in Italy.

1.4.2 CS in the Curriculum

According to the current national curricular recommendations, computing-related topics and digital technologies should pertain to two rather broad areas:

- A cross-disciplinary key citizenship *digital competence* area:[17] proficiency and critical attitude in the use of ICTs for work, life, communication; use of computers to retrieve, assess, retain, produce, present, share information as well as to cooperate through the Internet.

- A general *technology* subject area (grades 1–8) or a specific informatics/IT-related subject (grades 9–10 for some types of schools, to be taught by qualified teachers), which partly overlaps with the above area, but *may* also include some computer and/or robot programming.

Moreover, with regard to the basic competences at the end of primary and lower secondary education for the scientific-technological area, the national recommendations refer to some general awareness of the implications of using ICTs (for society, environment, health, and so on) and just add that "*whenever possible, students can be introduced to simple and flexible programming languages in order to develop a taste for the creation and for the accomplishment of projects* [...] *and in order to understand the relationships between source code and resulting behavior.*"[18]

However, the actual implementation of the curricular recommendations is, to a large extent, responsibility of each school, in accordance with the degree of autonomy introduced by the reform - an autonomy that may occasionally be exploited by self-motivated teachers to propose valuable initiatives also in informatics education. By contrast, what is improperly

[17] *Digital competence* is one of the seven broad areas listed in 2006/962/EC.
[18] http://www.indicazioninazionali.it/ (in Italian)

called "Informatics" was (and often still is) mostly taught as learning how to use ICT tools (e.g., using drawing programs or word processors).

1.4.3 Teacher Training

In Italy, lower and upper secondary teachers must hold a degree in the subject (or in a near subject, e.g., Math degree to teach CS) they aim to teach, and - only since recent years - some form of teaching qualification (e.g., a specific one year master).

By contrast, to become a pre-school and/or a primary school teacher in Italy, one currently has to get a 5-year (Combined Bachelor and Master) Degree in *Primary Teacher Education*. The course prepares students to become generalist teachers, by giving theoretical, methodological, and practical training to teach all subjects included in primary school (Italian, Math, History, Geography, English, Science, Technology, Sports, Music, to name the most important ones). Most of the teachers teach more than one subject in a class, sometimes both literary subjects and scientific/technical ones.

Because of the lack of Informatics in the national curriculum, its contents and teaching methods are not part of the Primary Teacher Education degree.

Much worse, that degree is mandatory to teach in primary schools in Italy only since 2002. Before that year, one could become a primary school teacher just with a High School Diploma specializing in Primary Teaching (again, preparing students to teach all the subjects), without even the need for a university degree. All teachers that got the Diploma before 2002 (more or less all teachers older than 40) are primary school teachers without a Primary Teacher Education degree. Due to low turnover, the vast majority of them belong to this category.

In both cases, apart from isolated professional development courses, most of the primary teachers and non-specialist secondary teachers did not receive any formal training neither in CS nor in CS teaching methods.

1.4.4 Recent Developments

In Italy, a recent school system reform [Italian Parliament, 2015] explicitly states that it is mandatory to develop student's digital skills, with particular care to the development of *computational thinking*. In a subsequent policy by the Italian Ministry of Education, University and Research [2016], the so-called *Italian National Plan for Digital Education*, a three-year plan of concrete actions *for setting up a comprehensive innovation strategy across Italy's school system and bringing it into the digital age*, was set up with the objective of *bringing computational thinking to all Primary Schools*.

Moreover, a recent addendum to the Italian National Recommendations for K-8 [Italian Ministry of Education, University and Research, 2018] adds *computational thinking* as a skill in which students must become fluent. The term *Informatics* (or synonyms) does not appear in the document.

The Ministry of University, Education, and Research (MIUR) and the National Interuniversity Consortium for Informatics (CINI – a consortium made up of all Italian research universities active in Informatics) agreed, in March 2014, to launch the project "*Programma il Futuro*"

("Program the Future") [Corradini, Lodi, and Nardelli, 2017a] to introduce informatics elements and activities in Italian schools, presented in the next section (1.4.5).

In 2017, a working group from CINI, in which the author of this dissertation is personally involved, authored a *Proposal for a national Informatics curriculum in the Italian school* [Nardelli et al., 2017; Forlizzi et al., 2018]. The proposal is described in Chapter 7 and fully reported in Appendix B.

1.4.5 The "Programma il Futuro" Project

The "Programma il Futuro" (PiF, from now on) project started from the school year 2014-15.

The initiative has been framed and presented in term of learning *computational thinking* as a key competence for modern education, to stress the importance of the cultural value of informatics more than its technical and technological aspects.

PiF project is based on the teaching material developed by Code.org. All Italian teachers are invited at the beginning of each school-year to use it in their classes, at least for one hour. Participation is optional, and any teacher is encouraged to engage in, whichever is her own teaching subject. This approach was chosen due to its scalability characteristics. By leveraging on-site teachers and well suited online teaching material, it is possible to faster bring to action a much larger number of students. In perspective, the project aims to facilitate the establishment of adequate informatics education in all school levels in Italy.

While targeting primarily teachers and students, the initiative has also addressed the adult population, according to the principle that education in fundamental concepts in informatics has both an intrinsic intellectual value and a practical role in understanding the IT basis of today's societal mechanisms. The project material is also used in adults' training centers, out-of-school education initiatives, and for self-learning, even by the elderlies.

PiF translated the textual material of all the tutorials (both online exercises and unplugged activities), paying particular attention to scientific precision and consistency.

PiF implemented a support website[19] to allow teachers building an Italian community of users. Following a carefully designed communication plan and an iterated development approach, the site provides a comprehensive guide for teachers.

In particular, in the *Percorsi* ("Paths") section offers, for each course and lesson of the courses proposed by Code.org, a detailed web page (in Italian) that explains concepts taught in that lesson and its general and specific learning objectives. Additionally, for each lesson, a video tutorial in Italian has been realized[20] to provide step-by-step guidance to any user towards successful completion of the activities.

In the last years, the project also proposed activities for upper secondary school students (e.g., building apps), paths for digital security and awareness, paths on computer architectures, creativity contests, and teachers' professional development courses.

According to May 2019 data, more than 6,400 schools, 31,400 teachers, 128,000 classes, 2,500,000 students participated in the project, doing an average of 15 hours of programming per student[21].

[19] http://www.programmailfuturo.it/

[20] The author of this thesis realized many of the concept explanations and video-tutorials.

[21] https://programmailfuturo.it/progetto/monitoraggio-del-progetto

1.5 Conclusions

Computer Science / Informatics is the scientific discipline that underpins the digital technologies that pervade our modern society. Understanding its scientific principles is a fundamental skill for every citizen. That is why many governments are introducing some elements of CS in the K-12 school curriculum, and, in the meanwhile, a lot of private and non-profit initiatives have emerged to involve kids in CS-related activities.

In Italy, some elements of CS are sparsely present in the K-12 curriculum, and some reforms are introducing "computational thinking" and "coding" as mandatory skills to be acquired. However, a comprehensive curriculum reform, formally bringing CS as a discipline into it, is still missing. Moreover, the vast majority of K-12 teachers are not trained to teach CS and, except for those with a specific CS background, they never studied the subject themselves at school: teacher training is, therefore, one of the most important steps to foster this introduction.

The Programma il Futuro project is encouraging Italian teachers to introduce elements of CS in schools, by supporting them - for example - with teaching materials translated from Code.org and a support website.

Chapter 2

Computational Thinking and Coding

The term *computational thinking* seems to have been firstly used in print by Seymour Papert [1980], and then was brought to the attention of the CS community by Jeannette Wing [2006].

From 2006, a considerable body of literature has been produced to search for a better definition of this concept, to provide tools and frameworks, to introduce and assess CT in K-12 education. For a review, see Grover and Pea [2013].

Even if there is no agreement between authors, a lot of proposed definitions stress the fact that CT is not only about technical methods and practices, but also about mental processes and transversal competences[1] like *creativity, collaboration, tolerance for ambiguity, resilience,* and more.

In this chapter, we review some of the most important definitions proposed in the last years, and propose a classification of the common elements that can be useful to better understand the misinterpretations of the concept.

We will then discuss different views about the extensive use of the word "coding" in the context of CS education.

2.1 Computational Thinking

The mathematician and computer scientist Seymour Papert (one of the creators of LOGO programming language - see 8.3.1, and the proponent of the *constructionist learning theory* - see 3.2) seems to be the first to have used in press the expression "computational thinking" [Papert, 1980, p. 182]. However, during his career, he made no attempt to define the concept precisely. Nevertheless, the original context in which that expression was used is still relevant today: for this reason, in Chapter 6, we present a historical analysis of that context, together with the history of the "idea" of CT in computer science.

In 2006 Jeannette Wing brought the expression "Computational Thinking" back to the

[1] Often referred also as *transversal skills*, *soft skills* or *key competences*, in the context of EU documents, in particular in the "*Personal, social and learning competence*," see for example http://data.consilium.europa.eu/doc/document/ST-5464-2018-ADD-2/EN/pdf

discussion [Wing, 2006], gaining a massive attention[2]. In that seminal article, Wing did not give a definition, but related the concept to the computer science discipline, stating "*Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science*". However, she did not confine the positive effects of learning this "skill" to professionals or scholars: Wing argued that "*thinking like computer scientists*" would be a benefit for everyone, in whatever profession involved.

Despite the vagueness of the proposal, Wing's position was taken as a trampoline for several initiatives to bring computer science into all levels of K-12 education, some of which have been described in Chapter 1.

2.2 Five Definitions of CT[3]

In these years, many definitions have been proposed. Juškevičienė and Dagienė [2018] schematized many of the definitions proposed from Papert's views up to 2017.

In this chapter, we focus on five of the most famous ones:

- the "Aho-Cuny-Snyder-Wing" definition of CT [Wing, 2011] (that builds on the informal definition in Wing [2006] and the philosophical discussion in Wing [2008]);
- the 2011 Operational Definition from International Society for Technology in Education (ISTE) and the Computer Science Teachers Association (CSTA) [ISTE and CSTA, 2011b];
- the definition proposed by Google in its collection of CT resources [Google, [n.d.]];
- the framework proposed by Brennan and Resnick about CT in Scratch [Brennan and Resnick, 2012];
- the definition from UK project Barefoot-Computing At School [Csizmadia et al., 2015].

Wing

Jeannette Wing informally defines CT as "thinking like computer scientists" [Wing, 2006] and then more formally as

> the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent. [Wing, 2011]

This definition is attributed to Jan Cuny, Larry Snyder, and Jeannette M. Wing, in an unpublished work from 2010: "Demystifying Computational Thinking for Non-Computer Scientists," referenced by Wing [2011] herself. Moreover, Wing says it was originated

[2]Currently (February 2020), the paper has more than 5800 citations, according to Google Scholar.

[3]This section is based on Corradini, Lodi, and Nardelli [2017b]

by a discussion with Alfred Aho, who provided a very similar definition, more focused on "algorithmic thinking":

> We consider computational thinking to be the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms. [Aho, 2011]

It is worth noticing that Aho stresses in particular the role played in this definition by the information processing agent, and that computational thinking should be based on clearly defined models of computation.

Wing also identifies characteristic elements of CT. In particular she states the most important elements are *abstraction* (the "mental" tool of computing) and *automation* (by using a computer, the "metal" tool of computer scientists): "*computing is the automation of our abstractions*" [Wing, 2008]. Wing [2011] recognizes important overlapping or inclusions between CT and other types of thinking: logical thinking, algorithmic thinking, parallel thinking, compositional reasoning, pattern matching, procedural thinking, and recursive thinking.

ISTE, CSTA, and Google

ISTE and CSTA propose an operational definition, targeting specifically K-12 educators. They define CT as

> a problem-solving process that includes (but is not limited to) the following characteristics:
>
> - Formulating problems in a way that enables us to use a computer and other tools to help solve them
> - Logically organizing and analyzing data
> - Representing data through abstractions such as models and simulations
> - Automating solutions through algorithmic thinking (a series of ordered steps)
> - Identifying, analyzing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources
> - Generalizing and transferring this problem-solving process to a wide variety of problems
>
> [ISTE and CSTA, 2011b]

Moreover they state that CT is "*supported and enhanced by a number of dispositions or attitudes*" that include

- Confidence in dealing with complexity
- Persistence in working with difficult problems

- Tolerance for ambiguity
- The ability to deal with open ended problems
- The ability to communicate and work with others to achieve a common goal or solution

[ISTE and CSTA, 2011b]

Finally they propose a CT vocabulary [ISTE and CSTA, 2011a], listing a set of CT terms with a brief definition/explanation: *Data Collection; Data Analysis; Data Representation; Problem Decomposition; Abstraction; Algorithms and Procedures; Automation; Simulation; Parallelization.*

Google assumes the same ISTE/CSTA definition but - instead of a vocabulary - lists and (re)defines a series of CT concepts [Google, [n.d.]], pointing out that they are *mental processes* or *tangible outcomes*: *Abstraction; Algorithm Design; Automation; Data Analysis; Data Collection; Data Representation; Decomposition; Parallelization; Pattern Generalization; Pattern Recognition; Simulation.*

CT with Scratch Framework

Brennan and Resnick [2012] present a *computational thinking framework*, to describe learning and development that take place when designing and programming interactive media with Scratch platform (see 8.3.6.1). They state CT involves three dimensions: *computational concepts* designers employ as they program: sequences, loops, parallelism, events, conditionals, operators, and data; *computational practices* designers develop as they program: being incremental and iterative, testing and debugging, reusing and remixing, and abstracting and modularizing; *computational perspectives* designers form about the world around them and about themselves: expressing, connecting, and questioning.

Computing At School

Csizmadia et al. [2015] assumes a Wing-like definition: CT is "*learning to think in ways which allow us, as humans, to solve problems more effectively and, when appropriate, use computers to help us do so*" and then states it involves six concepts (*Logic; Algorithms; Decomposition; Patterns; Abstraction; Evaluation*) and five approaches (*Tinkering; Creating; Debugging; Persevering; Collaborating*).

2.2.1 Common Aspects

We compared the CT elements found in the analyzed definitions.

Those who give a precise definition agree on the fact that CT is a **way of thinking** (thought process) for **problem solving**. They all somehow specify that it is not just problem solving: the formulation and the solution of the problem must be expressed in a way that allows a **processing agent** (a human or a machine) to carry it out.

Apart from the general statement, all definitions list some constitutive elements of CT. These elements are of very different kinds (from thinking habits to specific programming

concepts), and many authors group them in categories, but there is no universal agreement on the classification.

We classified all the elements into four categories. For each category, we list the elements, trying to summarize all aspects stated in the analyzed definitions.

1. **Mental processes**: mental strategies useful to solve problems.
    - *Algorithmic thinking*: use algorithmic thinking [Wing, 2008; 2011; ISTE and CSTA, 2011b] to design a sequence of ordered step (instructions) to solve a problem, achieve a goal or perform a task [ISTE and CSTA, 2011a; Google, [n.d.]; Brennan and Resnick, 2012; Csizmadia et al., 2015].
    - *Logical thinking*: use logical thinking [Wing, 2011] and reasoning to make sense of things, establish and check facts [Csizmadia et al., 2015].
    - *Problem Decomposition*: split a complex problem in simpler subproblems to solve it more easily [ISTE and CSTA, 2011a; Google, [n.d.]; Csizmadia et al., 2015]; modularize [Brennan and Resnick, 2012]; use compositional reasoning [Wing, 2008].
    - *Abstraction*: get rid of useless details to focus on relevant information or ideas [Wing, 2011; ISTE and CSTA, 2011a; Google, [n.d.]; Brennan and Resnick, 2012; Csizmadia et al., 2015].
    - *Pattern recognition*: discover and use regularities in data and problems [Google, [n.d.]; Csizmadia et al., 2015; Wing, 2011].
    - *Generalization*: use discovered similarities to make predictions or to solve more general problems [Google, [n.d.]; Csizmadia et al., 2015].
2. **Methods**: operational approaches widely used by computer scientists.
    - *Automation*: automate the solutions [Wing, 2008; ISTE and CSTA, 2011b]; use a computer or a machine to do repetitive tasks [ISTE and CSTA, 2011a; Google, [n.d.]].
    - *Data Collection, Analysis and Representation*: gather information/data, make sense of them by finding patterns, represent them properly [ISTE and CSTA, 2011a; Google, [n.d.]]; store, retrieve and update values [Brennan and Resnick, 2012].
    - *Parallelization*: carry out tasks simultaneously to reach a common goal [ISTE and CSTA, 2011a; Google, [n.d.]; Brennan and Resnick, 2012], use parallel thinking [Wing, 2011].
    - *Simulation*: represent data and (real world) processes through models [ISTE and CSTA, 2011b; Google, [n.d.]], run experiments on models [ISTE and CSTA, 2011a].
    - *Evaluation*: implement and analyze solutions [ISTE and CSTA, 2011b] to judge them [Csizmadia et al., 2015], in particular for what concerns effectiveness, and efficiency in terms of time and resources [ISTE and CSTA, 2011b].

- *Programming*: use some common concepts in programming (eg. loops, events, conditionals, mathematical and logical operators [Brennan and Resnick, 2012]).

3. **Practices**: typical practices used in the implementation of computing machinery based solutions.

   - *Experimenting, iterating, tinkering*: in iterative and incremental software development, one develops a project with repeated iterations of a design-build-test cycle, incrementally building the final result [Brennan and Resnick, 2012]; tinkering means trying things out using a trial and error process, learning by playing, exploring, and experimenting [Csizmadia et al., 2015].
   - *Test and debug*: verify that solutions work by trying them out [Brennan and Resnick, 2012]; find and solve problems (bugs) in a solution/ program [Csizmadia et al., 2015].
   - *Reuse and remix*: build your solution on existing code, projects, ideas [Brennan and Resnick, 2012].

4. **Transversal skills**: general ways of seeing and operating in the world fostered by thinking like computer scientists; useful life skills that *can enhance* thinking like a computer scientist.

   - *Create*: design and build things [Csizmadia et al., 2015], use computation to be creative and express yourself [Brennan and Resnick, 2012].
   - *Communicate and collaborate*: connect with others and work together to create something with a common goal and to ensure a better solution [ISTE and CSTA, 2011b; Brennan and Resnick, 2012; Csizmadia et al., 2015].
   - *Reflect, learn, meta-reflect*: use computation to reflect and understand computational aspects of the world [Brennan and Resnick, 2012].
   - *Be tolerant for ambiguity*: deal with non-well specified and open-ended, real-world problems [ISTE and CSTA, 2011b].
   - *Be persistent when dealing with complex problems*: be confident in working with difficult or complex problems [ISTE and CSTA, 2011b], persevering, being determined, resilient and tenacious [Csizmadia et al., 2015].

The classification proposed in the previous section was conducted at the end of 2016, as a theoretical basis for the analysis of teachers' conception about CT, described in Chapter 11.

In the meanwhile, other definitions and classifications based on literature review have been proposed. We report the most significant in Appendix A. We are confident in stating that there is a very high overlap between the definitions/classifications reported in appendix and our proposed categories.

2.3 Misconceptions about CT definitions

Note that many of the cited elements in the previous classification are broad and general. This led to some critiques [e.g., Hemmendinger, 2010]: some of these concepts are not exclusively associated with CS, but taught in other disciplines (e.g., Math and Sciences) or are general skills that children have been learning for a long time before the birth of CS. Anyway, many authors argue that computing features extend and differentiate these elements from other domains [Grover and Pea, 2013], and provides some characteristic problem-solving methods (e.g., the possibility to effectively execute a solution/a model/an abstraction by running an implementation of its algorithm [Martini, 2012]).

Voogt et al. [2015] recognize, in some of the mentioned definitions, a tension between "*the 'core' qualities of CT versus certain more 'peripheral' qualities*". The latter highly overlap with what we called "transversal skills," and we agree with Voogt et al. [2015] that "*including a broad list of this nature runs the risk of diluting the idea of CT, blurring and making it indistinct from other 21st century skills*". As we will argue in Chapter 6, we believe CT must be understood *inside* the discipline of computing.

By contrast, as CT movement has grown in educational contexts, and many unverified claims about the effects of learning CT/CS has emerged (e.g., that it will *automatically* transfer to thinking logically, better problem solving in every aspect of life, developing perseverance, getting better results in math and science, and so on [Lewis, 2017]). Most of these claims are not supported by research, and "*appear in blog posts, opinion pieces, and other 'grey literature'*" [Duncan, 2019].

In the light of these comments, we believe that the classification we proposed in 2.2.1 is a good tool to frame the misconceptions about CT and CS in K-12 education [Denning et al., 2017; Denning, 2017]. We anticipate here some leitmotivs that the reader will recognize in Parts II, III, and IV.

1. **Mental processes** are, on the surface, shared with other disciplines, but should be understood and experienced as CS specific. First of all, definitions are clear in stressing on the *computational* (rather than *general*) nature of *problem solving*. Next, as diSessa [2018] points out, *abstraction*, one of the core CT concepts according to Wing, has different nuances in different disciplines (e.g., between Math and Physics). Moreover, as we will see in Chapter 4, it is not clear if such general skills exist, are teachable or transferable: for example, *decomposition* - one of the widely highlighted CT skill [Guzdial, 2019a] - seems not to be easy transferable between contexts (see 4.4).

2. **Methods** must be experienced in the context of learning CS: they can be effectively introduced without computers, but need a clear link and experience through programming (see 8.4).

3. **Practices** are shared with other disciplines and activities, but computers provide powerful tools to "concretely experiment with" (see Papert's constructionist ideas - Section 3.2 and Chapter 6).

4. **Transversal competences** like perseverance and tolerance for ambiguity are useful for learning a difficult topic like CS, but including it in the definition may cause people to think CT is mainly about these competencies (which are easily recognized by generalist educators - see for example Chapters 10 and 11), and, even worse, get the wrong direction of the implication: they think these skills are automatically fostered by learning CT (which is yet to be proven: see for example our research in Chapter 12).

Nonspecialist teachers that most probably never studied CS in their schooling or training (see 1.4.3) may tend to stick to some "general versions" of mental processes and transversal competences totally unrelated to CS.

2.4 Coding[4]

The word "coding" is becoming more and more a buzzword, especially in CS K-12 education. As mentioned in 1.3, there are a lot of initiatives, like Code.org and its *Hour of Code*, the EU Code Week, CoderDojo, Code Clubs, websites like CodeAvengers, CodeMonkey, CodeCombat, and so on, aiming to teach students to "code." These initiatives are spreading and, since many governments are introducing *computational thinking* (CT) or *computer science* (CS) in school curricula, the term is used in many schools as well, especially referring to introductory programming activities.

According to some authors, unlike the expression "computational thinking," that may sound abstract and pretentious, and "programming" that seems to recall a boring professional activity [Ben-Ari, 2015], the expression "coding" can capture the interest of students, and "*also provides an element of mystery (there are hints of a secret code), and achievement (cracking the code)*" [Duncan et al., 2014]. There is a tendency in the media to use the term "coding" extensively, as noted - among others - by Armoni [2016] and Sentance [2018], when talking, for example, about "coding education." Some media observers have noted that "coding" is nowadays often used to denote a "*more playful and non-intimidating description of programming for beginners*" [Prottsman, 2015].

In popular culture, the term has also come to be used on the one hand as a synonym for the entire software development process, and on the other as a means to speak about what needs to be taught in school. This overlooks both the fact that coding/programming is only a part of the software development process, and that software development is only one of the important areas of computer science [Bell, 2016].

The expression "coding" is currently invested with excessive importance [Bell, 2016], and this may lead to the wrong idea that its value is greater than the CS scientific concepts themselves. This is particularly relevant since, in this initial phase of the introduction of CS in schools, many teachers self-train themselves and look for ideas and materials in the media, given training materials and professional development initiatives are scarcely available.

Anecdotally, we spotted these tendencies in Italy too. Moreover, in Italian, the term "programming" (translated as *programmazione*) has a very broad meaning (e.g., it is used for "schedules" like "movie show-times") and, in the context of schools, it is used to indicate

[4]This section is based on Corradini, Lodi, and Nardelli [2018b]

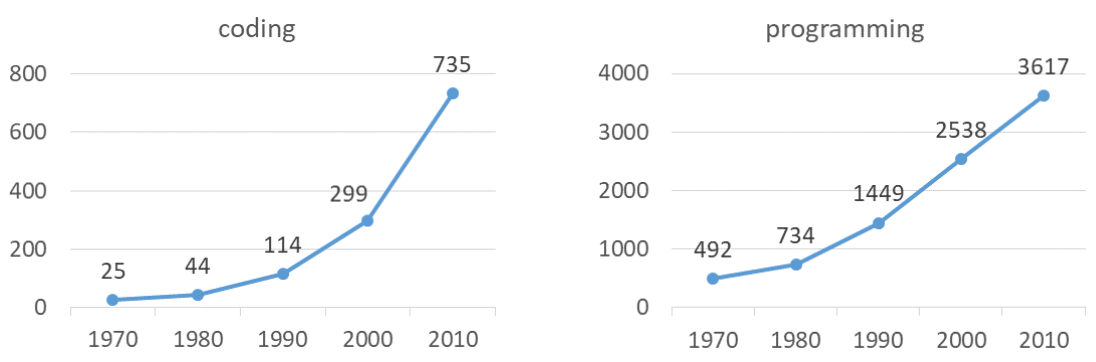


Figure 2.1: Growth of search hits for terms *coding* and *programming* in ACM SIGCSE publications. Source: ACM Digital Library search results (Aug. 7th, 2018).

"didactic planning"[5]. Furthermore, in Italy, there is a trend to incorporate "as they are" foreign terms indicating new concepts, rather than finding a corresponding Italian word. In facts, the term "coding" was explicitly used (untranslated) in the mentioned plan launched in 2015 by the Italian government and aiming at rendering Italian schools more digital (Italian National Plan for Digital Education - *Piano Nazionale Scuola Digitale* [Italian Ministry of Education, University and Research, 2016]), and widely reported in communication actions related to it.

Moreover, as anticipated in Chapter 1, a recent addendum to the Italian National Recommendations for K-8 [Italian Ministry of Education, University and Research, 2018] states that "coding" (again, untranslated) and "computational thinking" (*pensiero computazionale*) are skills in which students must become fluent, without mentioning CS.

A search (conducted in August 2018) in the ACM Digital Library, restricted to the SIGCSE publications, returns 1,186 hits for the search term "coding" compared to the 8,674 hits for "programming." However, the former shows exponential growth from the 1970s, while the latter just a linear growth (Fig. 2.1).

There is no agreement in the CSEd community about the relationship between coding and programming. In fact, some authors use the two terms interchangeably as synonyms or state both (e.g., they write "programming/coding"). On the other side, a few authors did not consider them as equivalent and analyzed their difference. They agree the term "coding" is more and more used in the tech business world as a jargon synonym word for programming, understood by other professionals [Armoni, 2016] (e.g., asking for "coders" instead of "programmers" in job offers). They also observe that on the one hand the term "coding" has a broader meaning in CS (e.g., in cryptography or in information theory), and on the other hand, it is often used to indicate the stage of software development when programs are actually written [Duncan et al., 2014; Armoni, 2016; Barendsen et al., 2015]. In other words, "coding" is considered as a narrower concept excluding important phases like analysis,

[5]For example in primary schools teachers meet weekly to do an "hour of programming," namely to agree on the content of lessons of the week.

design, testing, debugging [Bell, 2016].

We think that, while in the scientific community it is clear that *coding* and *programming* have a strict relation, and that they are only tools to teach what matters (i.e., CS core concepts - see Chapter 6), the confusion induced by this "coding mania" in the media can be very harmful. In fact, in our culture CS has been plagued almost since its teenage years by a lot of misconceptions [Denning et al., 2017], and it took decades to eradicate the limiting idea that CS is only programming [Armoni, 2016; Wing, 2006].

2.5 Conclusions

The expression "computational thinking" has become a buzzword related to the introduction of CS in K-12 education. Although it had already been used in the 80s by Papert (see Chapter 6), it started to be massively used in CSEd after being re-proposed by Wing [2006].

Many authors tried to define CT: despite being quite different, the most famous definitions share many characteristics. All agree CT is a form of thinking for solving problems by expressing the solution in a way that can be automatically carried out by an (external) processing agent. We identified four categories of CT constitutive elements proposed by authors: mental processes, methods, practices, and transversal skills. We argue that this classification can be useful to frame misconceptions about CT: problems related to each category will be analyzed in the rest of this thesis.

The diffusion of the term "coding" (used untranslated in Italian) to identify a playful introduction to programming, which is having a broad diffusion in schools and is present also in Italian ministerial documents, can generate misconceptions: while CS scholars understand that "coding" or "programming" is only (one of) the means to teach CS big ideas, non-trained teachers can easily be confused on what the goal is. We will explore this problem in Part III.

Chapter 3

Pedagogy

In this chapter, we briefly review the relevant learning paradigms common to many learning theories. We will give an overview of psychological, philosophical, and mostly pedagogical principles of the four major paradigms (behaviorism, cognitivism, constructivism, and humanism).

We will focus in particular on *constructivism*, which is currently the most researched and debated, and is at the basis of two specific theories or frameworks: *constructionism* and *creative learning*, that we will discuss in detail, as they form the basis of relevant ideas presented in this thesis.

3.1 Learning Theories and Paradigms

Learning theories describe *how people learn*. Philosophy, psychology, and pedagogy produced several different learning theories across centuries (see Fig. 3.1). It is out of the scope of this thesis to review them in detail: for a synthetic dictionary about learning theories, organized in different paradigms, see Leonard [2002].

Despite the usage of the terms varies a lot between authors, we will start by defining *learning paradigms* (used in this work as a synonym of *educational paradigms* or *paradigms of education*).

Definition (Learning paradigms [Leonard, 2002])**.** *Learning paradigms are classifications of learning theories into schools based upon their most dominant traits.*

Leonard [2002] classifies learning theories into four main paradigms: *behaviorism*, *cognitivism*, *constructivism* and *humanism*. Others distinguish more broadly between *instructivism* and *constructivism*.

3.1.1 Behaviorism

Behaviorist psychology dominated the first half of the 20th century, focusing on the exclusive study of *observable behavior* of human beings, while excluding internal mental states.

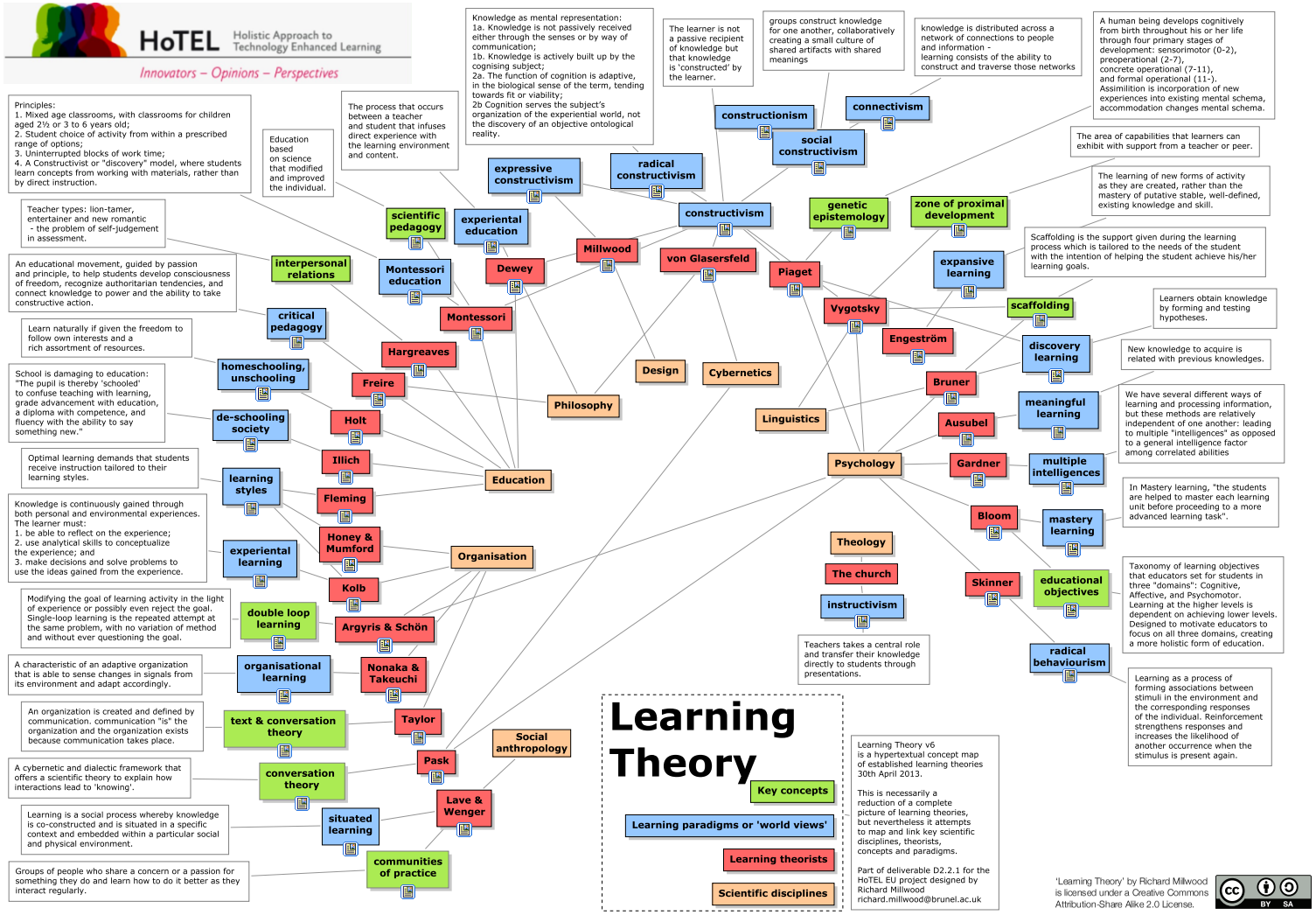


Figure 3.1: Learning Theory

For an interactive version of the map, see https://cmapspublic3.ihmc.us/rid=1LNV3H2J9-HWSVMQ-13LH/

By Richard Millwood (2013)

Basis of the theory are *classical conditioning*, from Pavlov's studies on stimulus-response in dogs, and Skinner's *operant conditioning*, studying how to influence behavior with positive or negative reinforces.

In this paradigm, learning is seen as guided and shaped through the "*process of conditioning by [...] sequences of stimuli, responses, feedback, and reinforcement*" [Falkner and Sheard, 2019, p. 446], in order to change students' form or frequency of observable behavior. Internal mental operations like *attention* and *memory* are not considered.

According to Baker et al. [2019], key elements of behaviourst educational paradigm are the following.

- *Purpose of education is to shape desirable behaviour*
- *Learning is change in form or frequency of observable behaviour*
- *Emphasis is on producing observable and measurable outcomes*
- *Learners are blank slates who passively receive information*
- *Instruction is repetitive; teachers shape behaviour through reinforcement*

From a philosophical point of view, this paradigm has an *objectivist* view of knowledge: there is an ontological reality, and its structure can be taught and learned.

Behaviorism is the primary influencer of *instructivist* teaching, that is focused on **transmission** of knowledge, and so focused "*on the structure and presentation of the learning material rather than* [on] *the learners who act as recipients of the instruction*" [Falkner and Sheard, 2019, p. 447].

Some learning theories belonging to this paradigm are:

- *programmed instruction*, proposed by Skinner, focused on the teaching of small units of knowledge (to give specific positive reinforcements), in an individualized way (through the prophetic "teaching machines");
- *mastery learning*, that heirs from the previous, with learning objectives divided into learning units of increasing difficulty, in which students work at their own pace.

Behaviourism still has a considerable influence in the present school system and in teachers' conceptions (for example, Martinez et al. [2001] found that "*the majority of these teachers shares traditional metaphors depicting teaching and learning as transmission of knowledge*").

Moreover, behaviorist ideas like reinforcement are the basis of the modern idea of gamification [Rice, 2012].

3.1.2 Cognitivism

The psychological paradigm of cognitivism was born in the 1950s, with the idea of going over behaviorism and study also people's internal mental states.

Cognitivists study human internal information processing in analogy with information processing within computers, viewing "*the brain as a processor of information and thought as a form of computation*" [Robins et al., 2019, p. 245].

Cognitivism elaborated many theories that still hold the stage nowadays, for example, multi-store memory model, cognitive load, mental schemas and models, prior conceptions and misconceptions, and so on.

In this paradigm, learning is achieved "*through a variety of learning strategies that depend on the type of learning outcomes desired, reflecting associated cognitive processes* [...including] *memorization, drill and practice, deduction, and induction*" [Falkner and Sheard, 2019, pp. 446,447].

Philosophically, cognitivism ranges from an *objectivist* epistemology to a more *constructivist* one. The former, while taking into account internal cognitive processes, still holds the instructivist idea of knowledge transmitted from the outside. The latter thinks subjects construct their own knowledge based on their previous experiences and social context, as we will see in the next subsection.

According to Baker et al. [2019], key elements of cognitivist educational paradigm are the following.

- *Purpose of education is for learners to remember* [memory] *and apply* [transfer, see Chapter 4] *information*
- *Learning is a change symbolic mental constructions (or schema)*
- *Emphasis is on structuring, organizing, and sequencing information in the mind*
- *Learners are information processors*
- *Teachers facilitate optimal processing*

The learning theories based on this approach focus on reorganizing information to foster its processing and acquisition (e.g., with chunking, reducing cognitive load, and applying learning taxonomies like Bloom's taxonomy).

3.1.3 Constructivism

Constructivism is an educational paradigm born during the 20th century to contrast the instructivist idea of knowledge transmitted by teachers and received by students. Philosophically it takes the moves from the idea that objective reality does not exist or is not knowable. To constructivists, there is no "true" knowledge out there, but knowledge is instead constructed personally or within groups and contexts. Unlike instructionism, it is student-centered and advocates active methods of learning.

Constructivism is a popular buzzword in education and comes in thousands of shapes and degrees of radicalism. According to Sorva [2012, p. 77], two central ideas appear in most of the constructivist views of learning:

1. *The learning of something new builds on the learner's existing knowledge and interest that learners bring into the context.*
2. *Learning is the construction of new understandings through the interaction of the existing knowledge and new experience.*

Among the many forms of constructivism, we can distinguish between cognitive constructivism and social constructivism.

Key theorists of **cognitive constructivism** are Jean Piaget, with his theories on stages of child development, Jerome Bruner, proposing ideas of discovery learning, and John Dewey, advocating for hands-on learning and experimental education. According to Baker et al. [2019], key elements of cognitive constructivism are the following.

- *Purpose of education is to enable learners to create new knowledge*
- *Learning is the process of constructing meaning*
- *Emphasis is on active discovery*
- *Learners actively construct new knowledge, building on what they already know and past experiences*
- *Teachers facilitate discovery by providing necessary resources*

A key figure of **social constructivism** is Lev Vygotsky, theorizing the construction of knowledge as social interaction, and known for the idea of zones of proximal development (ZPD): potential levels of development that children can reach with assistance. According to Baker et al. [2019], key elements of social constructivism are the following.

- *Purpose of education is for learners to co-create knowledge*
- *Learning is co-constructing knowledge and norms through social interaction*
- *The emphasis is on human relationships, learning through participation (activity) in social contexts (communities)*
- *Learners are active participants*
- *Teachers facilitate social interactions and collaborative work*

Most of the contemporary learning theories and approaches, making their way into schooling and certainly driving much of the educational research (and debate, as we will see), can be included in the constructivist paradigm. Among others, it is worth mentioning the following, in which it will be easy to recognize more cognitive or more social traits. For a general review, with CS contextualization, see Fincher and Robins [2019, chapters 8, 9, 15] and Sorva [2012].

- *Active learning* is a broad set of practices in which students actively do and reflect in order to learn. Nowadays often is used as a general term for learning approaches inspired by constructivism.

- *Cognitive apprenticeship* and *situated learning* try to replicate the traditional relationship between master and apprentice by working in authentic contexts and provide individualized guidance.

- The theory of *productive failure* tries to leverage on the idea of "learning by mistakes" by firstly asking students to solve ill-structured problems beyond their abilities, and then providing them canonical solutions or necessary knowledge, asking them to integrate

their solutions to consolidate knowledge. Recent studies "*have shown similar levels of procedural fluency to direct instruction, in addition to significantly better gains in terms of conceptual knowledge and knowledge transfer*" [Margulieux et al., 2019, p. 216]. For discussions on transfer and learning approaches, see Section 4.3.1.

- In *problem-based learning (PBL)*, students are faced with a realistic problem (e.g., a medical case) and have to learn the concepts to solve it autonomously.
- *Inquiry-based learning* is similar to PBL but draws on the scientific method: students create their own questions, obtain data, analyze them, and try to explain them by connecting with theoretical knowledge.
- *Community of practice* are "*communities of peers gathering to share and develop knowledge in a common context*" [Falkner and Sheard, 2019, p. 448].
- *Collaborative learning* is a set of practices in which students collaborate during learning, for example, in project-based teamwork on open-ended problems.
- *Cooperative learning* is a specific kind of collaboration in which students work in small groups, with defined roles, and each one is responsible for the learning of the whole group.

3.1.3.1 Critics to the Constructivist Paradigm

Constructivist theories have been (and still are) heavily criticized.

Some critics are related to philosophical underpinnings, like relativism. In the context of this thesis, we are not concerned with philosophy of education disputes. We like the approach that Matthews [1997] defines "*pedagogical constructivism*": concentrating "*solely on pedagogy, and improved classroom practices, and* [calling constructivist] *anything which is pupil-centered, engaging, questioning, and progressive*" [Matthews, 1997, p. 8].

Other critics are related to research methodologies (e.g., design-based research) used by constructivists, accused of scarce scientific rigor. Constructivists respond that "*scientific rigor is not worth research that is conducted in sterile environments (i.e., labs) that are fundamentally different from the authentic environments (e.g., classrooms) in which the research will be applied*" [Margulieux et al., 2019, pp. 212-213].

From a pedagogical point of view, the most emblematic critics come from Kirschner, Sweller, and Clark [2006], in the article titled "*Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching*". Their thesis - the dominance of direct instruction on minimally guided instruction - is based, in a very cognitivist fashion, on results and hypothesis in cognitive sciences, for example, that expertise comes from thousands of examples stored in long-term memory (see discussion in Section 4.2), or that novices benefit from a lot of guidance (e.g., studying worked examples), or that working on ill-structured authentic examples requires too much cognitive load from novices, not used to handle so much new information in their working memory.

Many authors responded to these critics, and a constructive debate was built in the book edited by Tobias and Duffy [2009].

We will use the cognitive thesis of Kirschner, Sweller, and others to question the possibility of teaching higher-order thinking skills (Section 4.2). We, however, agree with Taber [2012], stating that it is a very simplistic view to label all constructivist approaches as "minimally guided." Surely, some radical constructivists follow a vision (often referred to as *discovery learning*), where students are left entirely free to explore and learn with minimal guidance. Many other constructivists, however, agree that

> the level of teacher guidance (a) is determined for particular learning activities by considering the learners and the material to be learn; (b) shifts across sequences of teaching and learning episodes, and includes potential for highly structured guidance, as well as more exploratory activities. [Taber, 2012, p. 39]

To Taber, this is in line with the idea of learner-centered teaching but far from minimal guidance. He is suggesting to vary the degrees of **scaffolding**. The idea is predominant in the work of Vygotsky [e.g. 1978], but was formalized by Wood, Bruner, and Ross [1976]. They define scaffolding as a process "*that enables a child or novice to solve a task or achieve a goal that would be beyond his unassisted efforts.* [It requires] *controlling those elements of the task that are initially beyond the learner's capability, thus permitting him to concentrate upon and complete only those elements that are within his range of competence*" [Wood et al., 1976, p. 90].

Taber recognizes that, if learning is viewed as an iterative interaction between individual mental models and experience, the total absence of external guidance can only exacerbate the development of a very idiosyncratic view of the world. Society and education have precisely the role of mitigating this drift, to facilitate the development of world views that meet some general consensus. This is why he advocates a kind of constructivism that is student-centered but teacher-directed, which he calls **optimally guided instruction**.

> The aim of constructivist teaching then is not to provide 'direct' instruction, or 'minimal' instruction, but *optimum* levels of instruction. Constructivist pedagogy therefore involves shifts between periods of teacher presentation and exposition, and periods when students engage with a range of individual and particularly group-work, some of which may seem quite open-ended. However, even during these periods, the teacher's role in monitoring and supporting is fundamental. [Taber, 2012, p. 57, emphasis as in original]

If not otherwise specified, we will follow this kind of constructivism in the research work described in this dissertation.

In the context of CS, Guzdial [2019b] similarly proposes a balance between projects and direct instructions. This does not mean to fall back to transmissive and traditional lecture approaches: he also advocates for active learning techniques.

> Students in computing should work on projects. It's authentic, it's motivating, and there are likely a wide range of benefits. But if you want to gain specific skills,

> e.g., you want to achieve learning objectives, teach those directly. Don't just assign a big project and hope that they learn the right things there. If you want to see specific improvement in specific areas, teach those. So sure, assign projects — but in balance. Meet the students' needs AND give them opportunities to practice project skills.
>
> And when you teach explicitly: Always, ALWAYS, ALWAYS use active learning techniques like peer instruction. It's simply unethical to lecture without active learning. [Guzdial, 2019b]

3.1.4 Humanism

Humanism considers learning as the growth of the human being as a whole. According to Baker et al. [2019], its key elements are the following.

- *Purpose of education is for learners to progress towards autonomy and the realization of one's full potential*
- *Learning is personal growth*
- *Emphasis is on human freedom, dignity and potential*
- *Learners are in control of their education, learner-centered*
- *Teachers provide students with a non-threatening environment so that they will feel secure to learn*

A famous humanist learning theory is Montessori education.

3.2 Constructionism

Jean Piaget was one of the mentors of a South African mathematician and computer scientist: Seymour Papert.

As we have seen, Piaget is considered one of the fathers of constructivism. He theorized that the way we acquire knowledge determines how much it is valid for us, advocating the "*use of active methods which give broad scope to the spontaneous research of the child or adolescent and require that every new truth to be learned be rediscovered or at least reconstructed by the student, and not simply imparted to him*" [Piaget, 1973, p. 15].

Piaget's *constructivism* will be transformed by Papert into his own learning theory, **constructionism**. It shares with constructivism the idea of active building of knowledge through experience; it adds that learning is especially effective when the learner is consciously engaged in the active construction of objects[1] meaningful to her. To construct these objects, she needs *building materials* (concrete or abstract). Papert [1980, p. vi], for example, states that as a child he was obsessed with gears. He always used mental models about how gears work as a tool to understand the world, and even complex mathematical concepts like differential equations. Piaget distinguished between "concrete" and "formal" thinking,

[1] Public entities, "*whether it's a sand castle on the beach or a theory of the universe*" [Papert and Harel, 1991].

the first already present at the age of first grade and consolidated afterward, the second which does not appear until, say, age 12. Papert argues that "*the computer can concretize (and personalize) the formal*", thus allowing "*to shift the boundary separating concrete and formal.*" For him, anything can be easily understood, if it can be assimilated to the collection of mental models already present in the learner's mind. This is why one needs "objects to think with," the building bricks of the personal (construction of) knowledge. In the choice of these materials, however, there is not only a cognitive aspect - for the constructionist, at play there is always a fundamental affective component. Papert himself says he was *in love* with gears [Papert, 1980, p. viii].

Every student will be obsessed by something different, and here comes the power of computers, their protean ability to simulate and execute every other model, so that they may bring to everyone the building materials she loves most. Finally, the environment where learning happens is also fundamental. Computers can create a world where, for instance, you "speak mathematical language" (Papert called it Mathland) - like you learn a foreign language by living in a foreign country, you learn deep mathematical concepts by experimenting, and having concrete, practical experiences in Mathland. To provide these building materials and create these worlds (called "microworlds"), Papert and colleagues designed the educational language LOGO, presented in Section 8.3.1.

3.2.1 Situating Constructionism

In a remarkably consistent constructionist style, Papert does not want to define *constructionism*: we have to construct it by ourselves through studying examples [Papert and Harel, 1991].

What Papert noticed looking kids learning in very different contexts and disciplines is that some kids are perfectly comfortable the traditional "planning" or "top-down" approaches, while others are following more of a "bricolage"[2] or "bottom-up" approach, without a predefined plan.

Another difference is the so-called *closeness to objects*: "*some people prefer ways of thinking that keep them close to physical things, while others use abstract and formal means to distance themselves form concrete material*" [Papert and Harel, 1991].

Constructionism aims to take into account both styles and levels of closeness, not only the "top-down" and "abstract" ones, traditionally favored by schools.

3.2.2 Constructionism vs. Instructionism

During a video-speech to an educators conference in Japan in the 1980s, Papert [80s] discussed the contrasting concepts of *constructionism* and *instructionism*. Note the analogy of the terms with the traditionally contrasting paradigms of *instructivism* and *constructivism* - however, Papert's versions are more focused on the relationship between computers and education. In facts, constructionism and instructionism are different views of educational innovation:

[2]Papert already used the word "tinkering," most common today, as a synonym.

- *instructionism* is focused on improving *teaching*: computers do the teaching (computer-aided instruction);
- *constructionism* is focused on improving *learning*: "*Giving children good things to do so that they can learn by doing much better than they could before*".

In this talk, and - as we will see in much of his work - Papert is concerned with "*how technology can change the way that children learn mathematics*". In his view, technology is helpful in giving kids new things to do so that they can make something interesting to them with Math, in the same way that engineers, scientists, bankers use Math in the real world. It means using "*mathematics* constructively *to construct something*".

Papert's educational views and objectives will be further discussed in the historical analysis in Chapter 6.

3.3 Creative Learning[3]

Mitchel Resnick is a Papert's former student and now "LEGO Papert Professor" at MIT. His research group at MIT Media Lab is called "Lifelong Kindergarten": this is because he thinks that, "*instead of making kindergarten more like the rest of school, we need to make the rest of school – indeed, the rest of life – more like kindergarten*", in which "spontaneous" bottom-up learning takes place, although with the support of teachers [Resnick, 2007; 2017a]. Resnick gives a clarifying example of a possible kindergarten learning scenario.

> Two children might start playing with wooden blocks; over time, they build a collection of towers. A classmate sees the towers and starts pushing his toy car between them. But the towers are too close together, so the children start moving the towers further apart to make room for the cars. In the process, one of the towers falls down. After a brief argument over who was at fault, they start talking about how to build a taller and stronger tower. The teacher shows them pictures of real-world skyscrapers, and they notice that the bottoms of the buildings are wider than the tops. So they decide to rebuild their block tower with a wider base than before. [Resnick, 2007].

This happens more easily among kindergarten students because they need simple materials like bricks or pencils; much more complicated would be for older students, who have to learn more and more complex and abstract concepts.

As already predicted by Papert, however, the availability of these materials can be provided by the virtual environments that can be created with computers and suitable programming languages, such as LOGO (see Section 8.3.1), and Scratch (see Section 8.3.6.1), developed by Resnick's group. Scratch allows kids to learn by experimenting with *conceptual* blocks - representing programming instructions - just like the children in the previous example did with *physical* blocks.

[3] Part of this section is adapted from Lodi [2018c].

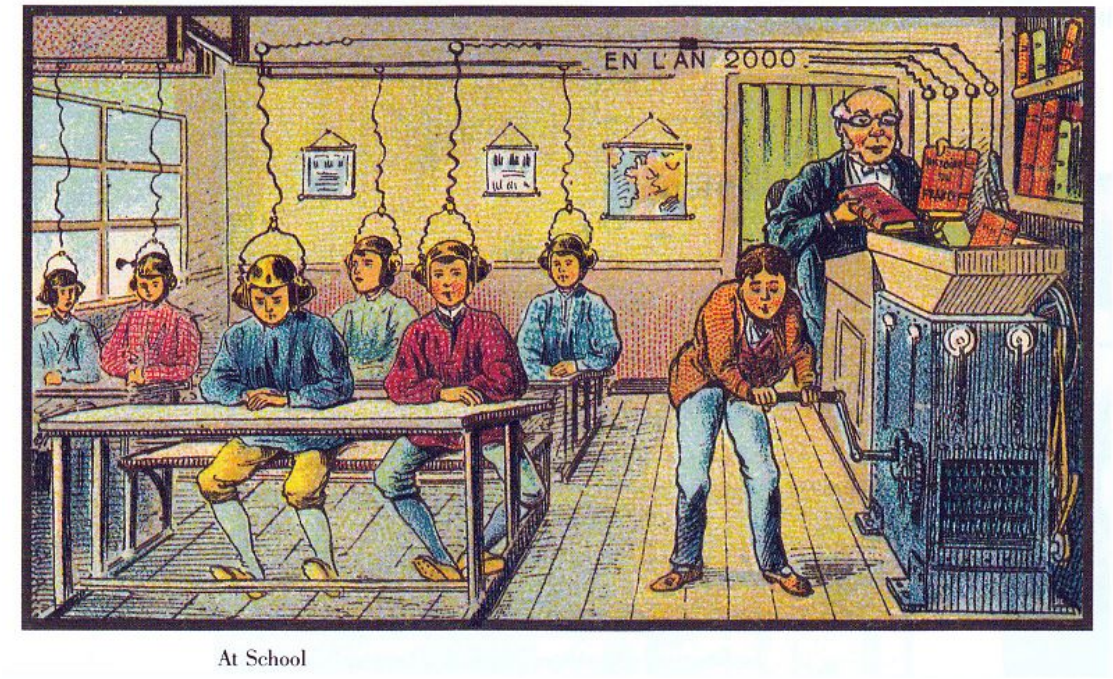


Figure 3.2: "At school": year 2000 imagined in 1900
By Jean Marc Cote (if 1901) or Villemard (if 1910)
Public domain, via Wikimedia Commons

Resnick claims that the current distance between kindergarten and school models is also due to the fact that society does not yet fully recognize the importance of teaching and learning *creative thinking*.

Let us discuss school and creativity by starting with a picture. A French illustrator tried to imagine, at the beginning of 1900, how it would be to live in the year 2000. Among the prints, he presented a hypothetical class in which students were instructed through a piece of machinery - connected to their brains - in which the teacher inserted the books (Fig. 3.2).

The disturbing scenario reminds us of both Skinner's programmed instruction, and current tendency to introduce computers and electronic devices in schools, but then to still use them for instructivist and transmissive teaching only.

Also, the image connects to an observation made among others by Papert [1993]. Imagine a teacher from 1900 teleported in today's class: she would probably understand very well the interactions that happen and could even conduct a lesson. The same does not hold for other professionals: if, for example, a surgeon from 1900 entered a modern operating theater would probably understand nothing of what is happening.

According to many observers, today's school is still modeled for a society that was in great need of employees: workers who should have been able and precise in performing mechanical and repetitive tasks imparted to them. However, this model is increasingly anachronistic for today's society: robots will increasingly automate the mechanical and repetitive tasks. Moreover, even jobs that today require a human "mental" component will be replaced by

more accurate artificial intelligence. What does - and will do more and more - the difference is no longer the knowledge of a series of notions or techniques, but rather the ability to be creative, innovative, the ability to adapt to a constantly changing world: a "liquid" society. If once the problem was the access to information, now we experience the opposite problem: we can access loads of information, and we seem to lack what would instead be decisive, namely critical sense and the ability to understand, to elaborate (also with the help of ICTs), and use this information for our own purposes.

For all these reasons, Resnick [2014] proposes a model called **creative learning**. According to Resnick, creative learning is based on the so-called "4 Ps". To learn creatively, one should do the following.

- Work on **projects**: it is important to ensure that students work actively on projects, rather than on exercises, that are often artificial and disconnected from reality, while a project is usually linked to something real and concrete. You can learn by testing new ideas, realizing prototypes, improving them, to obtain a "finished" product (which, however, can always be improved). Working on projects also allows us to learn *while* we use a tool, even a new one, rather than following the traditional but artificial direction of the "first I learn, then I do / use."

- Collaborate with **peers**: it is important to ensure that students collaborate, exchange ideas, build new things based on the work of their peers. Furthermore, the work of relevant figures is studied in many fields: the source code - or visual blocks - of programs made by others may represent the "literature" of computer science, and studying them can be valuable.

- Follow own interests and **passions**: if students are engaged in activities related to what they are passionate about, or they consider useful or important, they will concentrate more, and for a longer time, they will not discourage if they encounter difficulties, and ultimately learn more authentically and profoundly. Papert said "*Education has very little to do with explanation, it has to do with engagement, with falling in love with the material.*" [Resnick, 2017b].

- Accept challenges and **play**: children do not only play for fun, but above all for learning: it is important to stimulate students to playfully experiment with new ideas, take risks, proceed by trial and error, explore their own limits and potential. Resnick claims that instead of talking about "edutainment" (education + entertainment), which are two passive concepts, it is better to talk about "learn and play," which are two active concepts.

Following these principles, the construction of a creative artifact follows an iterative process [Resnick, 2007; 2017a], called "creative learning spiral" (see Resnick [2017a, p. 11]).

According to this model, when you learn by creating something (e.g., a computer program) you **imagine** what you want to do, **create** a project based on this idea, **play** with your creation, **share** your idea and your creation with others, **reflect** on the experience and feedback received from others, and all this leads you to **imagine** new ideas, new functionalities, new improvements for your project, or new projects. The process is iterated many times.

3.4 Conclusions

Philosophy, psychology, and pedagogy proposed several learning theories - describing how people learn. We analyzed, in particular, the three major paradigms in which we can categorize them: behaviorism, cognitivism, constructivism.

Behaviorism is associated with the traditional instructivist approach of knowledge transmission: its influence is still clearly visible in the current school system, and its ideas are present in a vast majority of teachers.

Cognitivism shifted the focus from the behavior of students to their mental processes, ranging between an instructivist approach to a more constructivist one. Its focus is on the reorganization of information according to cognitive studies to foster learning.

Constructivism is focused on learners: knowledge is believed to be built personally (cognitive constructivism) or socially (social constructivism) by learners rather than being transmitted to them. This paradigm is highly criticized, especially in its more radical forms, advocating minimal guidance during learning. We will stick to a more moderate view, the so-called "optimally guided instruction," giving students the optimal level of scaffolding needed to construct their knowledge actively.

A specific constructivist learning theory, constructionism, was originated by the mathematician and computer scientist Seymour Papert, who highlighted the vast potentials of computers and programming as tools to provide the (cognitive and affective) building materials useful to facilitate the construction of knowledge, even abstract one like the mathematical one.

The passive, instructionist, use of computers is also contrasted by the creative learning framework, proposed by Mitch Resnick, focusing on the use of programming as a medium of self-expression and of learning through creating artifacts, derived from the kindergartners.

Chapter 4

Transfer

In the context of this CS in K-12 movement, some misconceptions are spreading: claims state that studying CS can automatically improve students' results in other disciplines, or teach them domain-general problem-solving skills (e.g., abstraction, pattern recognition, problem decomposition, logical reasoning) or influence cognitive and affective aspects (e.g., motivation, self-efficacy, resilience, creativity).

Educational research warns against automatic transfer and agrees that transfer is difficult, especially between far domains and especially if activities are not explicitly designed for it. Moreover, different teaching approaches and tools can influence transfer.

We will review the critical literature on transfer, with particular focus on "higher-order thinking skills," and then we will have a closer look at the (long-dated) history of the relationship between CS Education and transfer.

4.1 Transfer of Learning

In the context of this thesis, we assume this definition.

Definition (Transfer of Learning [Ambrose et al., 2010])**.** *The application of skills (or knowledge, strategies, approaches, or habits) learned in one context to a novel context.*

One can argue that transfer is one of - or even *the* - final goal of education and schooling, hoping students will apply what they learn in different contexts (new problems in the same course, new courses, in their life outside school, at work, and so on) [Ambrose et al., 2010; National Research Council, 2000].

There are many forms of transfer, that we will summarize here[1]. First of all, we distinguish between [Robins et al., 2019]:

- **spontaneous transfer**, where "*the student recognizes the similarities between what they already know and the knowledge required to solve the new problem without help or guidance*";

[1]For comprehensive reviews, on which we mostly draw upon, see [National Research Council, 2000; Robins et al., 2019].

- **guided transfer**, where "*students can [...] be guided [...] by pointing out the similarities between old and new problems or asking the student to find similarities*".

Obviously, spontaneous transfer is the most desirable, but, as we will see, educators erroneously think that it happens automatically, while usually this is not true, especially for novices that do not have an in-depth knowledge of one domain in the first place, and struggle to apply it to different contexts [Robins et al., 2019]. Already in 1920, Alfred North Whitehead noted that students acquired what he called *inert* or even *brittle* knowledge: they used it to pass the exam, but not outside the school context [Guzdial, 2010].

Transfer is especially difficult when it is *far*. The "distance" of transfer is, in fact, an important factor in determining if and how it happens. Following the taxonomy proposed by Barnett and Ceci [2002], we can distinguish[2]:

- **Knowledge domain transfer**, relating to disciplines. It has been traditionally divided between *near* and *far*.
  - *Near transfer* happens inside a discipline.
    It can be *isomorphic* (between different instances of the same problem), *contextual* (different problem context, same problem solving procedure), or *procedural* (same type of problem, different solving procedure).
  - *Far transfer* happens between disciplines.
    It can have *several shared elements*, so the solving procedure can be used in both cases, or with *few shared elements*, that imply only analogical application (e.g. *sequence* in programming and in writing stories).
- **Physical context transfer**, if it happens among different learning spaces (that influence learning).
- **Temporal context transfer**, if it happens among time (it is what we call **retention**).
- **Functional context transfer**, if it happens among different purposes (e.g. passing the exam vs. delivering an actual project).
- **Social context transfer** if it happens between different social contexts (e.g. individual vs. group work).
- **Modality transfer** if it happens between different medium of instruction or application (e.g. same algorithm in *unplugged* and *plugged* activities).

All the forms of transfer presented so far are desirable, and often called **positive transfer**. However, transfer can also be **negative**, when previous knowledge can damage performance in a new context.

As we will see in Section 4.4, CS Education has been interested in transfer for a long time, and in particular, in *near transfer* inside CS. In this work, however, we are more interested in *far transfer*, between CS and the so-called *higher order thinking skills* (HOTS).

[2] As summarized by Robins et al. [2019].

Definition (Higher Order Thinking [Lewis and Smith, 1993])**.** *Higher order thinking occurs when a person takes new information and information stored in memory and interrelates and/or rearranges and extends this information to achieve a purpose or find possible answers in perplexing situations.*

Higher-order thinking skills include "*problem solving, critical thinking, creative thinking, and decision making.*" [Lewis and Smith, 1993].

In the next section, we highlight some important moments in the history of transfer and HOTS, relevant for our thesis.

4.2 Transfer and HOTS

The idea of transfer (from Math to other disciplines) can be found as early as in Plato, that, in his *Republic* (written in the 4th century BC), stated that

> [...] those who have a natural talent for calculation are generally quick at every other kind of knowledge; and even the dull, if they have had an arithmetical training, although they may derive no other advantage from it, always become much quicker than they would otherwise have been. [Plato, 1982]

This idea was still prevalent at the beginning of the 20th century. This "formal discipline" or "mental discipline" theory stated that learning difficult subjects like Latin, Greek or Math would have broader effects, like the development of domain-general skills [National Research Council, 2000], or, as Stanic [1986] put it, "*improves one's thinking and helps one learn to overcome difficult obstacles*". The reader will notice immediately that these ideas strongly resembles some of the most popular arguments for the introduction of CT (effects on general problem solving and on perseverance - see 2.3).

The same strong parallel can be found between Wing's arguments [Wing, 2006] and the ideas of Young [1906, p. 17], stating that "*the facts of mathematics, important and valuable as they are, are not the strongest justification for the study of the subject by all pupils. Still more important than the subject matter of mathematics is the fact that it exemplifies most typically, clearly and simply certain modes of thought which are of the utmost importance to everyone*".

However, the pioneering studies from Thorndike and colleagues [Thorndike and Woodworth, 1901; Thorndike, 1924] largely questioned the validity of this theory. In particular, Thorndike [1924, p. 98] concluded that "*the intellectual values of studies should be determined largely by the special information habits, interests, attitudes, and ideals which they demonstrably produce. The expectation of any large differences in general improvement of the mind from one study rather than another seems doomed to disappointment*".

These studies, of course, did not close the debate and were conducted in a behaviorist fashion, without taking into account any of the learner characteristics. In fact, in the following decades, two monumental works gave new strength to the HOTS movement, with particular focus on Math: the first is "How to solve it: a new aspect of mathematical method" by Pólya [1945] and the second is "Human Problem Solving," by Newell and Simon [1972]. As observed

by diSessa [2018], these works were followed by many rigorous studies about *disciplinary* problem solving, and by a lot of less rigorous "*curricular and commercial work purporting to teach domain-general problem solving skills*".

Experts in Math Problem Solving, like Schoenfeld [1985], recognize in the *multiplicity* the main obstacle in learning problem solving. To explain it with Thorndike:

> It is misleading to speak of sense discrimination, attention, memory, observation, accuracy, quickness, etc., as multitudinous separate individual functions are referred to by any one of these words. These functions may have little in common. There is no reason to suppose that any general change occurs corresponding to the words' improvement of the attention,' or 'of the power of observation,' or 'of accuracy.' [Thorndike and Woodworth, 1901, p. 249]

As diSessa [2018] concludes, "*the single most powerful element in gaining mathematical power is to understand deeply the concepts that the field deploys*".

Evidence that HOTS do not easily transfer was also found in more recent years. For example, Brown [1992] discovered that HOTS and metacognition are hard to teach and do not transfer outside the experimental context:

> [...] it turns out not to be easy to train a learner to be strategic, to select cognitive activities intelligently, to plan, to monitor, to be cognitively vigilant, economical, and effective! And it is particularly difficult to do so in arbitrary contexts where a learner is attacking meaningless material for no purpose other than to please an experimenter. [...] [I]t is for this reason that the decontextualized approach to metacognitive training was largely unsuccessful [...].
>
> [Brown, 1992, p. 146]

The importance of particular disciplines in developing HOTS is also undermined by the review by Ceci [1991], which surprisingly found that IQ and cognitive abilities were influenced mainly by the quantity of education that students got, rather than on the specific kind of education. They concluded that "*[m]ost schools in developed nations, no matter how poor, may be sufficient to maintain and develop IQ and related cognitive abilities*" [Ceci, 1991, p. 717].

Moreover, the teachability of HOTS is debated: for example, in recent years, in the context of the debate about so-called "brain training" games and software: "*[...] people get better at the particular exercises [...]. But they are unlikely to transfer that learning to non-exercise contexts. Most surprisingly, they are unlikely to transfer that learning even though they are convinced that they do.*" [Guzdial, 2016]. Similarly, a recent meta-review found negative evidence that learning chess, music, or performing working memory training "*does not reliably enhance any skill beyond the skills they train*" [Sala and Gobet, 2017].

This may be explained by the radical thesis of Tricot and Sweller [2013], stating that domain-general problem solving is part of the so-called "biologically primary knowledge": it is so vital that "*we have been selected for our ability to acquire it*", and so it is not teachable because we have already acquired it. Authors use the fact that we struggle to find - through

controlled studies - an effective and teachable domain-general cognitive strategy as a clue of the impossibility of teaching them. In their view, what is teachable is only "biologically secondary knowledge," in particular:

- domain-specific knowledge: in many studies, the domain-specific knowledge held in long-term memory can be used as the primary predictor of performance and difference between novices and experts (e.g., chess masters remember thousands of configurations more than novices rather than having different problem-solving skills; air-traffic controllers can multitask only when talking about airplanes, not other unrelated pieces of information; ability to remember long sequences of numbers does not transfer to the same ability with strings, and so on);
- the possibility of using a domain-general strategy (already acquired naturally) to solve a specific problem (and so leverage the acquisition of secondary knowledge).

Tricot and Sweller [2013, p. 280] also provide a possible explanation of why many people focus on domain-general skills.

> At any given time, we are unaware of the huge amount of domain specific knowledge held in long-term memory. The only knowledge that we have direct access to and are conscious of must be held in working memory. [...It] tends to be an insignificant fraction of our total knowledge base. With access to so little of our knowledge base at any given time, it is easy to assume that domain-specific knowledge is relatively unimportant to performance. It may be difficult to comprehend the unimaginable amounts of organised information that can be held in long-term memory precisely because such a large amount of information is unimaginable [...:] we are likely to search for alternative explanations [...like] domain-general strategies.

4.3 Current Pedagogical Research on Transfer

We will now briefly review the most important findings of educational research about transfer, following National Research Council [2000] and Ambrose et al. [2010].

First of all, most of the research found that:

- transfer occurs neither often nor automatically;
- successful *far* transfer is less likely the more contexts are novel and different.

Reasons for this can be found in:

- *over-specificity* or *context dependance*: students associate knowledge only with the original context in which they learned it;
- *lack of deep knowledge*: students often learn what to do but not why (memorize without understanding), and so lack of a deep understanding/mastery of principles and structures of knowledge. Moreover, they are distracted by superficial features of knowledge.

However, the research found that transfer can be explicitly fostered, but we have to "**teach for transfer**":

> employ instructional strategies that reinforce a robust understanding of deep structures and underlying principles, provide sufficiently diverse contexts in which to apply these principles, and help students make appropriate connections between the knowledge and skills they possess and new contexts in which those skills apply. [Ambrose et al., 2010, pp. 111-112]

Unlike the research from the beginning of the 20th century, modern research focused not only on practice, but also on individual learner characteristics, pedagogical approaches, learner motivation. In particular, we should:

- provide students adequate time to learn, so they can explore underlying concepts and generate connections;
- ask students to study "worked examples" (examples with a detailed and explained solution): research showed that, when they are complete novices, it gives better results than actually solving problems (which is, by contrast, better when they are more competent);
- balance context dependant concrete activities with abstract cross-contextual knowledge;
- make students apply what they learned in different contexts;
- ask students to make structured comparisons of different problems, to recognize deep, meaningful features rather than superficial features;
- discuss with students conditions of applicability (context in which a skill is - or is not - applicable) and pros/cons of different methodologies, also confronting them with "what-if" scenarios;
- ask students to find contexts in which a given skill can be applied;
- ask students to abstract and extrapolate principles from different contexts and examples;
- help student self-monitor their own learning (meta-cognition) and engage in "deliberate practice," also by giving feedback on their learning process;
- give students small prompts (e.g., stimulating students by recalling a different exercise from which they can borrow similar strategies, being careful of not vanishing the educational purpose of the new task);
- encourage analogical reasoning, use visual representations and make students articulate causal relationships;
- view transfer as a dynamic process (e.g., increased speed in learning a new domain rather than immediate better performance in it);
- work to increase student's motivation (ideally the intrinsic one) and mindset (see Chapter 5), so that the initial deep learning is fostered and transfer is then easier.

4.3.1 Transfer and Learning Approaches

Transfer is also influenced by teaching approaches, as discussed by Robins et al. [2019, sec. 9.6]. First of all, abstract instruction seems to foster spontaneous transfer. However, it is tough for novices to grasp deep abstract knowledge, so they need surface details. However, this can be ineffective since novices tend to stick to surface details and so fail to transfer to similar situations that are only apparently different.

As seen, one of the problems with transfer is that students do not learn or do not remember much in the first place. Direct transmissive instruction, the traditional method in higher education and in many schools, is argued to be more effective, since students are explicitly told what they need to know [Kirschner et al., 2006]. However research shows that constructive methods (see Section 3.1.3) can "*lead to better retention in the long run*". Moreover, through the so-called *generation effect*, "*learners are more likely to activate relevant prior knowledge to solve problems if they constructed that knowledge while working on problems*" [Robins et al., 2019, pp. 254,255].

Unfortunately, constructivist approaches are not always suitable, especially for complete novices [Tobias and Duffy, 2009].

An idea is combining the approaches, by finding the right "time for telling": traditional "teaching by telling" can be extremely useful when done with the right timing, for example, after people already struggled with the problem on their own [Schwartz and Bransford, 1998]. This idea is shared with the theory of productive failure (see § 3.1.3).

4.4 Transfer and CS Education

As anticipated, transfer has been studied in the context of CSEd. From one side, it has been studied *inside* the discipline of computing (near transfer - e.g., transfer between different programming languages and environments or between different CS courses).

Many results showed *negative transfer*, finding, for example, that "*learning a second programming language was sometimes harder than learning the first* [...and] *that student's knowledge of computer science is tightly tied to the way that their first programming language looks and feels*". Moreover, the research found negative transfer "*from communication in a natural language to coding in a programming language*" [Guzdial, 2015].

In the specific context of CT, the positive transfer of "Computational Thinking Patterns" ("*abstracted programming patterns that are learned by students when they create games and can readily be used by students to model scientific phenomena*") has been found from programming games to creating science simulations [Basawapatna et al., 2011].

Another important debate is about the use of visual programming languages (see 8.3.6.1) in introductory learning. Current research seems to point out in the direction of positive transfer between visual and textual interfaces [Hundhausen et al., 2009] and between block-based and text-based languages [Weintrop and Wilensky, 2019].

Thirdly, an ongoing debate regards the transfer from unplugged activities to the "plugged" ones. We discussed the topic in subsection 8.4.3.

From the other side, a big focus has also been put on the idea that studying CS can

transfer to HOTS, which is our primary focus here. The debate dates back to as early as the 1980s, in the context of the educational language LOGO (see 8.3.1). Claims about LOGO "teaching how to think" were highly questioned by some rigorous studies from Pea and colleagues (e.g., Pea and Kurland [1984]), finding, for example, no correlation between studying LOGO and being better at planning, together with misconceptions about how some aspects of LOGO work (e.g., recursion). This went together with other studies (e.g., Cannara [1976]) finding in LOGO students typical programming misconceptions (e.g., the "superbug" or confusion about parameter binding - see 8.2.2), but also difficulties in decomposing complex problems into smaller ones.

This is in line with the first problem of transfer: student did not learn much programming, so had only superficial knowledge, hard to transfer.

These studies gained attention because of the debate between Pea and Papert itself [Papert, 1987]. Pea asked for scientific rigor, while Papert talked about a much bigger change in "computer culture" and what it could do to education, so deep that needed different experimental approaches than the "treatment model."

In the same paper, Papert also discussed the importance of the different *methodologies* in which a tool (LOGO) can be used. This is a hint on the thesis that Papert's main message was misunderstood, as we will extensively discuss in historical research described in Chapter 6.

Some research and reviews were conducted in the early 90s. Palumbo [1990] did an important meta-review, and found no significant connection between studying programming languages (LOGO, BASIC, Pascal) and problem solving. Similarly to previous research, he highlighted that students did not learn much programming, and that most of the experiment had design problems and did not put sufficient attention to problem-solving theory. He "*concluded that more advanced forms of transfer (far or generalized transfer) should not be expected in introductory courses, since typically there is no time to develop such skills. In other words, if curricula aim for the transfer of problem-solving skills to other domains, explicit time and effort should be put into it.*" [Blikstein and Moghadam, 2019, p. 71]. We see the same problems again: low expertise and failure to explicitly teach transfer.

Liao and Bright [1991] did a similar review, finding conflicting results, concluding however that "*computer programming has slightly positive effects on student cognitive outcomes.*"

Moreover, in the same years, encouraging results were obtained when the interventions were explicitly designed to provide initial learning and to transfer specific competencies [Palumbo and Reed, 1991; Klahr and Carver, 1988]. In particular, Klahr and Carver [1988], in the context of LOGO programming, were able to teach debugging explicitly and to transfer it to "buggy" situations outside programming (e.g., in wrong road directions or furniture arrangement in a room).

More recent studies confirmed that a lot of expertise is needed before transferring computer science skills to problem solving: Mark Guzdial affirms that "*[m]ost people do not teach programming for transfer, and if they did, they would not be able to cover as much of programming. I think it is a zero sum game: Teach for programming fluency or teach for transferable problem-solving skills. You cannot get both in the same time.*" [Blikstein, 2018, p. 18].

According to Guzdial [2015, p. 50], what *may* transfer from learning some programming

in K-12 are "*more fundamental computational ideas*"[3], like:

> Computers are powerful but limited. They don't have an intelligent being inside of them [...].
>
> Programs are sequential and literal, with every line of the program being executed in sequence [...] students tend to view programs as being a parallel process, a set of constraints or events [...].
>
> There are a handful of operations that computers can do. Understanding the primitive operations of the notional machine [...] is a key learning outcome of introductory computing courses. Computers can test data and make choices based on the result [...] can iterate over actions forever until stopped or until a condition is met [...].
>
> We have to transform our problems to make them understandable to a computer [...].

In the context of modern days wave for CT in K-12 education, some studies are trying to connect learning introductory CS and transfer to other school subjects or even the so-called "21st-century skills", highly overlapping with what we are referring to as HOTS.

However, previous research must be treasured: for example, Gick and Holyoak [1980] found that *problem decomposition*, one of the most highlighted aspects of CT [Guzdial, 2019a], is not easily transferrable. They described to students a situation where an army had to be divided into small groups to successfully attack a fortress; immediately after they asked the students how to attack a tumor with a laser without damaging healthy tissues. The vast majority of the students were not able to use the same approach (divide the laser in multiple weaker beams). They only managed to do so when explicitly prompted to think at the army example.

Scherer [2016] recognizes that research on transfer and CS is contradictory and outdated, and must be conducted in relation to modern learning tools and contexts, and with modern rigorous educational research techniques. Moreover, he is optimistic about the possibility that computer programming skills can be transferred to domain-general problem-solving skills. He, however, uses as an argument the similarity between the domains in terms of the "*processes of exploring and understanding, representing and formulating, planning and executing, and monitoring and reflecting*" [Scherer, 2016]. However, as extensively discussed in the previous sections, teachability and transferability of HOTS are highly debated.

At the moment, only preliminary and contradictory findings can be reported.

A study in the context of Code.org and interdisciplinary laboratories for primary students (and teacher PD), found that "*teacher completion of a higher percentage of 'extra' Code.org CS lessons was significantly associated with*" improvement of students in Literacy, Math and Science. The subtitle of the document, "Preliminary Findings of an Exploratory Study" [Century et al., 2018] clearly indicates there is much more to analyze.

A study by BT and Ipsos-MORI [2016] on the English Barefoot project

[3]Part of what we will call *the natural sediment of disciplinary learning of computer science* in Chapter 6.

> found that primary teachers using the Barefoot resources reported that children using these resources were less "needy" and were developing computational thinking skills, thinking for themselves and building resilience. In this study, every teacher (100%) said that their pupils enjoy using technology in lessons. One in six (16%) teachers reported that pupils liked technology because it is creative, while one in five teachers felt that children liked using technology to problem solve and discover how the technology itself works (20 %) [...] 22% of the teachers reported that children found using technology fun and engaging and the same number that it was familiar to them from use at home. [...] There is less evidence that the Computing curriculum is effective in terms of skills and knowledge development. [Sentance and Waite, 2018, p. 107].

However, as Guzdial argues[4] "*[s]tudents might say (self-report) how they planned to use computing in their daily lives [...]. Self-report does not mean that they actually change their problem-solving behavior to use computing outside of a computing context.*" [Guzdial, 2015, p. 40]. As seen for "brain training" (§ 4.2), people can be convinced that it improves their general abilities, but it does not. This is not surprising: as we will see during this dissertation, educators, stakeholders, and even researchers tend to stick with transversal aspects of CT and forget the CS core concepts characterizing it.

Kazakoff et al. [2012] found, in a small sample of pre-kindergarten and kindergarten kids, a "*significant, positive impact on sequencing scores with just 1 week of working with robotics and programming*".

Arfé et al. [2020] found that following eight hours of Code.org activities not only improved kids ability to solve coding problems, "*but it also positively affected children's* [... executive functions:] *children exposed to Coding activities increased their planning time and accuracy and decreased the rate of inhibition errors.*" The greater effects (and retention) were found in tasks that were more similar to the Code.org mazes.

By contrast, Kalelioğlu [2015] explored the effects of Code.org on fourth-grade students, finding that "*programming in code.org site did not cause any differences in the reflective thinking skill towards problem solving*".

In the same vein, Duncan [2019], in her Ph.D. dissertation, confirms that her research

> [p]rovided evidence against the claim that by learning Computational Thinking skills through Computer Science and programming students will automatically transfer this learning to other areas. However, there is some evidence that the majority of students can transfer these skills if they are explicitly taught to apply Computational Thinking skills in different contexts, or if Computer Science and programming are taught in a cross-curricular way. [Duncan, 2019, p. 246]

Successful examples of **using** CS (and in particular programming) mainly as a tool to foster the learning of other subjects (e.g., Algebra [Schanzer et al., 2018] or Physics [diSessa, 2018]) are available: however, the amount of computer science learned by students is confined

[4] Referring to Cutts et al. [2011], where college students self-reported about the usefulness of - and transfer from - an Alice course in their daily lives or work.

to what they strictly need to use in the main subject they are learning [Guzdial, 2015]. This, however, is less related to transfer and more with *computational literacy* [diSessa, 2018].

All the reasoning about transfer from CS to general problem solving applies to an even furthest form of transfer: from CS to more affective aspects like resilience, grit, and growth mindset [Lewis, 2017].

However, Pugh and Bergin [2006], in a review on motivation and transfer, suggest that motivation could, indirectly, influence transfer. In fact, motivation can lead to deeper initial learning, to deliberate attempts to transfer concepts, and to persistence in tasks. More directly, the transfer is correlated with mastery goals (typical of a growth mindset - see § 5.1) more than performance goals are, and with and self-efficacy. Authors, however, recognize that extensive research is needed.

Transfer from CS/CT to a growth mindset (Ch. 5) has been investigated by the author of this dissertation in two different contexts: results will be presented in Chapters 12 and 13.

4.4.1 A Recent Meta-Review

The issue is far from decided: in a comprehensive, very recent meta-review, Scherer et al. [2019] analyzed the literature on computer programming and transfer. They found the reviewed studies showed evidence for a strong effect for near transfer (effects of learning programming on programming skills) and moderate effect on far transfer (effects of learning programming on contexts that are different from programming and may require different skills or strategies).

Regarding far transfer, they found that it "*seems to exist for computer programming, yet not for all cognitive skills and not to the same extent*" and it is "*more likely to occur in situations that require cognitive skills close to programming*." In particular (remembering that, overall, far transfer was moderate), they found higher effects on creative thinking (in particular in the sub-dimension of originality) and mathematical skills (but in particular on geometrical concepts after the LOGO language was used). Lower effects ("*possibly due to a larger degree of domain-generality*") were found on metacognition, spatial skills (again possibly explained by the focus on geometry or movements) and reasoning (which, in most cases, is assessed with tests that include problem solving, intelligence, and memory). An even smaller effect was found on school achievement, and no effect was found on literacy (measured as reading comprehension and writing).

However, the authors recognized several methodological issues in the reviewed studies, advocating the need for more specific and accurate research.

4.5 Conclusions

We analyzed the crucial idea of transfer of learning, classifying it through different dimensions. We highlighted, in particular, the difference between near transfer, inside a discipline, and far transfer, that happens between different disciplines or contexts.

We then focused on the specific problem of transfer from difficult subjects like Math to higher order thinking skills (HOTS) like domain-general problem solving, a century-old

debate. It is still open, but, in general, one has to be cautious in making exaggerated claims and should focus more on the disciplinary content and specific way of thinking a subject can teach, rather than on more general skills.

According to current educational research, transfer is difficult for novices because they tend to associate knowledge with the original context in which they learned it, and because they lack in-depth knowledge and general principles to apply to other contexts. However, transfer is achievable if teaching activities are specifically designed to do so, with several strategies useful, for example, to help novices handle the cognitive load and to help them recognize the relationship between different pieces of knowledge.

The effect of different learning approaches (instructivism versus constructivism) on transfer is also still debated between experts.

For what regards CS, near transfer has been studied (e.g., how to move from one language or paradigm to another). Also, the debate on transfer from CS to HOTS dates back to the introduction of LOGO in the 80s. The question is still open and discussed, with scattered studies showing contrasting results, but, in general, confirming that near transfer is easier than far transfer, and the furthest the skill is from CS, the harder the transfer seems to be.

Chapter 5

Implicit theories

To non-computer scientists, learning to program may appear as a too challenging goal, achievable only from those having an innate talent for programming, the so-called *geek gene*, an idea shared by many CS educators as well [Ahadi and Lister, 2013]. Despite recent research seems to refute its existence [e.g., Patitsas et al., 2016], the results are far from conclusive [Robins, 2019]. Moreover, CS is subject to stereotypes based, among other things, on gender, social status, or ethnicity. For example, Lewis et al. [2016] found that some people identify computer scientists with *singularly focused*, *asocial*, *competitive*, *male figures*.

In this chapter, we will focus particularly on mindset theory, concerning the effects of personal ideas that people hold about their own intellectual abilities. This cognitive theory has recently gained much attention among educators around the world.

We will review significant findings related to the theory and its implementation in schools, also acknowledging doubts and critics that some researchers raised about it, and discuss possible explanations of the current research status. We will then focus on the small set of studies focusing on the relation between CS and mindset theory, acknowledging they are contradictory and inconclusive.

5.1 Mindset Theory

Students and teachers have different personal ideas (*implicit theories*) about their intellectual abilities. Some believe that their intelligence is a fixed trait they born with (like eye color or height when adult), and they cannot do much to change it: they have an *entity theory of intelligence*, otherwise stated a *fixed mindset*. Some others believe instead that intelligence is a malleable trait that can be developed with study and deliberate effort (like muscles can be trained): they have an *incremental theory of intelligence*, also called a *growth mindset*.

Mindsets theory (as is nowadays worldwide known - more formally called *implicit theories* or *self-theories* by its proponents, and also *lay theories* or *naive theories* by others) is a fundamental result of three decades of research by Stanford psychologist Carol Dweck and her team [Dweck and Leggett, 1988; Dweck, 1999; 2017b].

This is not a new idea (it can be found, among others, in John Dewey and Albert Bandura), but the mindset "branding" is currently widely popular among educators (at least

in the English speaking countries) [Hendrick, 2019].

5.1.1 Mindset Effects

Studies show that students' mindset can influence motivation, reaction to challenges, and academic results [Blackwell et al., 2007; Dweck and Yeager, 2019]. Growth mindset is positively correlated with grades and achievements, and can be useful to reduce gender disparities in STEM: girls with a growth mindset showed less susceptibility to the adverse effects of stereotypes about women and math [Good et al., 2012].

Students with different mindsets show different features and behaviors. Following Dweck and Leggett [1988] and Kaijanaho and Tirronen [2018], we schematize them in Table 5.1.

Table 5.1: Fixed and growth mindset features [Dweck and Leggett, 1988; Kaijanaho and Tirronen, 2018]

Fixed mindset	Growth mindset
performance goals: *appear* intelligent (e.g., by cheating) and hide weakness (e.g., by not asking questions)	**learning-oriented goals**: not afraid to ask and make errors, in order to learn
helpless responses to challenges: giving up or blaming the teacher for failure	**mastery-oriented responses**: greater effort and new strategies when facing challenges and setbacks
low value of effort, seen as a sign of lack of talent	**high value of effort**, seen as a sign of learning
seek of easy tasks and avoidance of difficult ones	**seek of challenging tasks** in order to learn
negative emotions and **decreased problem-solving performance** when facing difficulties	**neutral or positive emotions** and **neutral or increased problem-solving performance** when facing difficulties

5.1.2 Mindset Interventions

Dweck and colleagues showed that a growth mindset could be fostered with specific interventions, and in particular:

- explicitly teaching students about mindsets, brain plasticity and the idea that intelligence can be trained with effort (e.g., through readings, videos, discussions) [e.g., Blackwell et al., 2007; Yeager et al., 2016; 2019];
- praising process and effort (e.g., "You worked so hard and did a very good job") rather than person or talent ("Bravo! You must be very smart!"), and give constructive feedback rather than praising the person or being judgmental [e.g., Mueller and Dweck, 1998; Dweck, 2017b];

- portraying challenges, effort, and mistakes as highly valued [e.g., Dweck, 2008].

According to Boaler [2013; 2015], specific suggestions to stimulate a growth mindset in Math also include:

- giving rich open tasks, requiring effort and reasoning;
- teaching for patterns and connections;
- teaching creative and visual mathematics.

5.1.3 Teachers' Mindset

Teachers have a more significant impact on student learning than any other variable [Boaler, 2015]. In particular, teachers' conceptions are crucial: in an unpublished study [Good et al., 2007] described by Dweck [2008], adults were asked to behave as teachers - after a fixed or a growth mindset about Math had been taught them. The "growth" group was more supportive with students, giving encouragement and suggesting positive strategies to deal with problems; by contrast, the "fixed" group subjects gave students pure comfort and fixed messages (like "*Not everyone is a math person!*") and tended to effectively help boys significantly more than they did with girls.

Teachers' growth mindset is strongly necessary, but not sufficient to foster a growth mindset in their students [Sun, 2015]. Teachers must also create a growth mindset environment where growth messages are sent, and explicitly teach students new strategies to cope with failures and to master the material [Dweck, 2017b].

5.1.4 Critics and Responses

Mindset theory has attracted in very recent years a lot of critics[1].

First of all, researchers outside Dweck's group found it hard to replicate results, and questioned the efficacy of large scale mindset interventions. Some reports and reviews found no correlation [e.g., Gorard et al., 2012; Foliano et al., 2019] or even a slightly *negative* correlation [Bahník and Vranka, 2017] between students' mindset and achievement. Sisk et al. [2018] recently conducted two meta-analyses and found that, in general, there was very little effect on achievement from mindset interventions. However, "*students with low socioeconomic status or who are academically at risk might benefit from mind-set interventions*" [Sisk et al., 2018, p. 549]. Moreover, some statisticians are raising concerns about statistical methods used in original mindset studies (see discussion in Lynch [2018] and Hendrick [2019]). Dweck and her team are responding precisely to critics by re-analyzing data (see, for example, documents shared by Dweck [2017a]).

Moreover, Dweck acknowledges that correctly implementing her theories is harder than she thought, especially by new researchers. She also states that her theories have been misunderstood and misapplied. In the updated version of her famous book, Dweck [2017b, pp. 214-216] talks about *false growth mindset* - ways in which mindset has been misunderstood.

[1] For an overview, see Lynch [2018] and Hendrick [2019].

First of all, people think fostering a growth mindset is only about praising students' effort, even when they are not really trying hard. However, one should focus on praising the *process*, and teaching how to overcome failures: "*hard work, trying new strategies, and seeking input from others*". Dweck fears "*that the mindset concept will be used to make kids feel good when they are not learning - just like the failed self-esteem movement*", and warns against convincing students that "they can do anything." Even worse, teachers say they have a growth mindset, but behave in very fixed ways, for example, by "recursively" applying mindset theory and stating "*I can't teach this child. He has a fixed mindset*".

Other authors, however, think that the difficulties in implementing the theory in schools are caused, among other factors, by the intrinsically fixed nature of the traditional school system, that is still dominated by instructivist practices (see Subsection 3.1.1) and puts much importance on grades and performances rather than on the learning process [Hendrick, 2019]. For example, Kohn [2015] argues that is not enough to focus on the learning process, effort and ways to overcome difficulties, but also focus on *what* and *how* students are learning: otherwise we are only pushing students to put their effort into an obsolete transmissive system (Sect. 3.3). He is advocating for more meaningful learning, for which they have an intrinsic motivation (in line with a constructivist view, see 3.1.3).

Another hypothesis [Hendrick, 2019] is that *mindset* is often taught as a domain-general skill, whose teachability and transferability is however highly debated (as extensively discussed in Section 4.2). Dweck herself acknowledges that one can have different mindsets with respect to different disciplines or tasks [Dweck, 2017b].

5.2 Growth Mindset in Computer Science

As opposed to other scientific disciplines, only a few studies have been conducted on the relationship between an introductory computer science/programming course and growth mindset.

In a survey administered to CS faculty members of a US institution, more than three-quarters of them *disagreed* on the fact that "*Nearly everyone is capable of succeeding in the computer science curriculum if they work at it*" [Lewis, 2007]. Similarly, teaching tutors in a study by Cutts et al. [2010] "*were themselves displaying a strongly-fixed mindset*".

Murphy and Thomas [2008] argue that *a growth mindset can be particularly important in CS* education. For example, when programming, students are continually facing errors and challenges; female enrollment in CS is low, possibly due to fixed views about who can and who cannot succeed in CS; collaboration is fundamental in CS and is fostered by growth mindset students whose goal is to learn rather than to appear smart, and so on. Factors like repeated exposure to errors and stereotypes are recognized to potentially foster a fixed mindset, as has been suggested for math [Boaler, 2013]. Cutts et al. [2010] say that Carol Dweck herself describes CS as a discipline that requires a growth mindset, referring however to an unrelated paper. We believe the authors committed an "off-by-one" error in their references. Probably, they wanted to refer to Dweck [1999, p. 37], where she reports a conversation between two college students. She argues they had a growth mindset because they were considering majoring in CS despite having repeated the CS exam a few times without getting brilliant

grades.

As seen, Dweck [2017b] acknowledges that one can have different mindsets with respect to different areas or subjects. Therefore, we define **Computer Science (CS) mindset** as the *mindset with respect to computer science*.

Lewis et al. [2011], looking precisely at *mindset about CS ability* found that "*a student's mindset* [with respect to CS] *can affect self-assessment of ability and, ultimately, motivation to major in CS*" . Some studies considered more narrowly the *mindset with respect to programming.*

Only a bunch of studies have been conducted to concretely assess student's mindset before and after a programming course. Flanigan et al. [2015] analyzed (without intervention) changes in CS1 (CS-major, other STEM-Majors, but also Arts and Business Majors) students across the semester, finding a significant increase in fixed mindset and a significant decrease in growth mindset. Moreover, mindset only weakly predicted students' outcomes. In Kaijanaho and Tirronen [2018], late bachelor-level CS students' mindset was assessed, finding no correlations between their mindset and their grades. Apiola and Laakso [2019] surprisingly found a (weak) positive correlation between academic achievement and *fixed* mindset in first-year CS students.

Other authors tried interventions, in line with the ones suggested by mindset literature, to alter students' conceptions of intellectual ability. Simon et al. [2008] tried a small intervention in CS1 classes to change the mindset of students from CS Majors and Minors. It is an explicit intervention where students, after being taught about mindset theory, are asked to write a letter to younger students to transmit ideas learned. The study obtained mixed results; in general, there was no difference between intervention groups and control groups: both shifted toward a more fixed mindset (generally and with respect to programming) despite the intervention. Cutts et al. [2010] performed three structured interventions into an introductory programming course, combining an "explicitly teaching" intervention with a "growth mindset messages" intervention. The control group moved towards a more fixed mindset, while both experimental groups had significant improvement in growth mindset levels. However, only students receiving both kinds of interventions showed also a positive effect on their test scores - an effect not found in the group that only received the "explicitly teaching" intervention.

Scott and Ghinea [2014] found that beliefs about intelligence and programming aptitude form two distinct constructs in undergraduate software engineering students. Moreover, the mindset for programming aptitude had greater utility in predicting software development practice, and a follow-up survey showed that it became more fixed after a programming module.

Very recently, Gorson and O'Rourke [2019] found that programming mindset calculated through standard questionnaires is often misaligned with how students talked about intelligence or programming behavior during interviews. Often students hold both fixed and growth mindset ideas. Moreover, students showed undesirable ways to measure programming aptitude, e.g., solving problems quickly and without any errors. This indicates they hold *a fixed view of CS as a discipline* (a problem further discussed in Chapter 12).

These studies seem to show that studying CS can foster a *fixed* mindset, which is, of course, highly undesirable. This clearly contrasts with popular claims of growth mindset being

automatically fostered by studying CS (for a discussion, see Lewis [2017]). In Chapter 12, we will present results supporting skepticism about these optimistic claims.

By contrast, we think some intrinsic characteristics of CS (at least if taught as a creative subject: e.g., with open, real, authentic projects, iterative approach, debug, trial and error, collaboration rather than competition) can foster a growth mindset. Preliminary findings in this direction are presented in Chapter 13.

In other fields, similar results were found: during the first year of University, engineering students tend to move towards a fixed mindset. However, introducing open-ended engineering design projects into the curriculum may tend to lessen or eliminate the shift toward a fixed mindset [Reid and Ferguson, 2014]. In this direction, Loksa et al. [2016] designed and tested an intervention to explicitly teach problem solving for open-ended programming problems, in the context of a web app programming summer camp for high school students. They gave participants a method to visualize and monitor their progress through six problem-solving stages, on-demand prompts to reflect on their strategies while seeking help, and context-aware helps embedded in the editor. They found the intervention had positive effects on productivity, self-efficacy, independence, meta-cognition, and growth mindset (in particular, by contrast with the control group, intervention students did not shift towards a more fixed mindset).

Except for the very last reported study, which, however, is not focused primarily on growth mindset, all the other cited experiments were conducted among college students. We were not able to find any study investigating correlations between growth mindset and introductory CS / CT courses for K-12 education or teacher training / professional development.

5.3 Conclusions

Mindset theory is becoming quite popular among educators and researchers. It is a theory based on decades of studies by Carol Dweck's team. They showed that having a growth mindset (believing that intelligence can be trained, like muscles) is far more desirable than having a fixed mindset (believing intelligence is an immutable trait like eye color). Students and teachers with a growth mindset get better results, cope better with difficulties, are less subject to stereotypes. Mindset can be taught by specific interventions. However, the theory is criticized because it has been hard to replicate the results independently. Moreover, the theory tends to be oversimplified and misinterpreted by educators.

For what concerns CS, only a bunch of studies have been conducted, leading to mixed and inconclusive results. Moreover, it is accepted that one can have different mindsets with respect to different disciplines, so we considered in particular CS mindset, the mindset with respect to computer science. Research in other disciplines seems to indicate that constructivist approaches can be beneficial for fostering a growth mindset.

In part IV we try both to refute simplistic claims about automatic transfer between CS and GM, and to test the effects of creative computing on mindset.

Part II

History, Epistemology and Pedagogy

Chapter 6

Computational Thinking: from Papert to Wing[1]

In this chapter, after recalling the evolution of the idea of "computational thinking", that we can find alongside the birth of the discipline itself, we present its meaning in the work of Seymour Papert, who used for the first time the expression in his *Mindstorms*. For him, the technical aspect of "thinking like a computer scientist", which is the main content of Wing's use of the term, cannot be separated from the social, and affective dimension of building computational objects in an environment rich of computational principles and meaningful for the community.

We will argue that both Wing's and Papert's views were misinterpreted, but both carry fundamental aspects for the introduction of CS in K-12 education.

6.1 Introduction

As discussed in Chapter 2, "Computational thinking" (CT) is one of the buzzwords of the moment, in CS K-12 education. The modern (and long) wave of this expression started, as it is well known, with a seminal essay by Jeannette [Wing, 2006], that paved the way for a large number of studies, which have produced educational material, definitions, even assessment methods [Grover and Pea, 2013].

After much hype about the subject, Tedre and Denning [2016] produced a critical review of CT, also framing it in its historical context.

We argue that, in its modern perspective, CT should be understood *inside the discipline of computing*, as *the (scientific and cultural) substratum of the technical competences.*

Historically, however, the expression CT was used for the first time by Seymour Papert in 1980, with a different *nuance* of meaning, which should not be forgotten. The analysis of this sense of CT will be the subject of Section 6.3.

Before turning to Papert's CT, however, we summarise some of the earlier attempts to identify the concepts that are peculiar to computer science, and which are now covered

[1] This chapter is based on unpublished material written with my supervisor, Prof. Simone Martini.

under the umbrella of CT. We refer to the lucid work of Tedre and Denning [2016][2] for a comprehensive historical account and assessment.

6.2 Prehistory

The end of the fifties and the early sixties are the years in which the field of computing gradually builds its self-understanding as an autonomous discipline [Tedre, 2014]. This process goes hand in hand with the need to specify the traits and concepts distinguishing the new discipline from other sciences, like applied mathematics, or physics, or engineering. A first, important, process is the *linguistic shift* of the programming task [Nofre et al., 2014]. In the early days, programming was mainly a technological affair (strictly coupled to the technology of the different computers). The emergence (and the need) of computer-independent ("universal", in the terminology of the time) programming languages allowed the expression of *algorithms* in a machine neutral way, thus making algorithms and their properties amenable to a formal study. Programming languages themselves were treated as object of study - from the formal definition of their syntax [Backus, 1959; Backus et al., 1960], to the gradual emergence of a mathematical theory of computation [McCarthy, 1960; 1961], and, later, of a mathematical semantics [Naur, 1966; Floyd, 1967]. Bruno Latour, with genial insight, explains in this way the relationship between a new science and its language:

> No scientific discipline exists without first inventing a visual and written language which allows it to break with its confusing past. [Latour, 1986]

The availability of universal programming languages is felt as the opportunity for computing to evolves from its "obscure" past (made of mathematics, cybernetics, logic, physics, engineering, linguistics) and consciously presents itself as the science of algorithmic problem solving, for which the new languages are developed. Of course, this "founding language" should not be identified with a specific programming language. It is an early recognition that the contemporaneous presence of *different* specific languages (at various levels, with various purposes, with various targets) is an asset of the discipline, and that no language will work for all uses[3].

It is in this context that, not later than 1960, Alan Perlis uses the term *algorithmizing* ("*quantitative analysis of the way one does things*"), classifying it as "*part of the basic thought processes*" that "*everyone should learn* [...] *sooner or later*" [Katz, 1960]. For him "*students will have a chance to use computers better* [...] *by virtue of understanding them as general tools to be used in reasoning* [...] *rather than as devices to solve particular problems*". It is one of the earliest recognition of a specific, disciplinary approach to problem solving that would be the result of being exposed to, and having acquired, the competences of that new field, which at that same time struggled to be recognised as an autonomous

[2] Peter Denning has been writing several critical papers on CT and its hype [see, e.g., Denning, 2009; 2017; Denning et al., 2017; Denning and Tedre, 2019].

[3] See, for instance, Gorn [1963] and its insistence on the role of mechanical languages.

scientific discipline[4]. In one of the many attempts to describe this new science and its boundaries, George Forsythe (first Head of Computer Science at Stanford) comments on the educational value of computer science: "*The most valuable acquisitions in a scientific or technical education are the general-purpose mental tools which remain serviceable for a lifetime. I rate natural language and mathematics as the most important of these tools, and computer science as a third*" [Forsythe, 1968].

It is especially in the seventies that this line of thought comes to maturity - the idea that Computer Science provides *general thinking tools*, useful for everyone (recall Perlis' position). Marvin Minsky in his Turing award lecture [Minsky, 1970] has a long section ("*developed with Seymour Papert*"[5]) on mathematics education. The thesis is that the computer scientist has the role, the responsibility, and the competences to "*work out and communicate models of the process of education itself*". The very last statement of the paper is that the computer scientist "*is the proprietor of the concept of procedure, the secret educators have so long been seeking.*"

Edsger W. Dijkstra, again in a paper discussing the epistemological status of programming in comparison to mathematics [Dijkstra, 1974], observes that programming gives a unique opportunity to master the complexity of a system, which is handled through a "*hierarchical composition,*" where "*a single technology*" (that of programming languages of different levels of abstraction) encompasses all the levels of the hierarchy. It is this dealing with "*mastered complexity*" which "*gives programming as an intellectual activity some of its unique flavors.*" The "*programmer's agility with which he switches back and forth between various semantic levels*" is a sort of "*a mental zoom lens*".

Among the many other possible citations and quotes, we conclude with Donald Knuth (one of the stars at Stanford, recipient of the Turing award in 1974, at the age of 36), who is convinced "*of the pedagogic value of an algorithmic approach; it aids in the understanding of concepts of all kinds;*" "*a student who is properly trained in computer science is learning something which will implicitly help him cope with many other subjects*" [Knuth, 1974].

What quoted authors depicted is an *idea* of computational thinking. We propose to characterize it as *the natural sediment of disciplinary learning of computer science* - that which remains behind when all the technicalities and the definitions of the discipline are long forgotten[6].

[4] A struggle that was going to be long. The first *Computer Science* department of the US was established in 1962 at Purdue University, where Perlis had served in the computation center from 1951 to 1956. Samuel D. Conte, first Head of that department, will recall in a 1999 Computerworld magazine interview: "*Most scientists thought that using a computer was simply programming - that it didn't involve any deep scientific thought and that anyone could learn to program. So why have a degree? They thought computers were vocational vs. scientific in nature*" (quoted in Conte's obituary at Purdue University, 2002). Next computer science departments to be established would be those at the University of North Carolina at Chapel Hill, in 1964, and at Stanford in 1965. Still in 1967, Perlis, Newell and Simon (three Turing award recipients; Simon will also be a Nobel laureate in Economics) feel the need of a letter to Science [Newell et al., 1967] to argue "*why there is such a thing like computer science*". See also Knuth's reconstruction of the contribution of George Forsythe to this process [Knuth, 1972].

[5] And again, in the introduction: "*Papert's views pervade this essay.*"

[6] We refrain from a historical reconstruction of this folklore expression, often said of "culture". It appears in print at least in 1908, in Gaetano Salvemini's "Cos'è la cultura".

Moreover, it is easy to spot, already in these early characterizations, the (largely undocumented and unproven - see Section 4.2) claim that the cognitive skills gained through programming (or, more generally, through computer science techniques) *transfer* to other disciplines - from the already quoted Minsky [1970] and the related Feurzeig et al. [1970][7] for mathematics, to the far reaching Mayer et al. [1986] and Pea and Kurland [1984], for which see also the commentary criticism by Salomon [1984].

However, the impact of this process on actual reform of education or, more generally, on the cultural debate was modest. Computers and computer science were still mainly confined in scientific and engineering milieux, and in large corporations. A situation which did not change much when this anonymous thinking received for the first time its current name.

6.3 Papert's Computational Thinking, in Context

Seymour Papert seems to be the first to use in print the expression "computational thinking" [Papert, 1980, p. 182]. Contrary to Wing's use in 2006, however, this single occurrence of CT in *Mindstorms* is by no means an attempt to a definition: "*Their* [of people using computers for providing mathematically rich activities] *visions of how to integrate computational thinking into everyday life was insufficiently developed.*" It is used *en passant*, after many other "computational" *something*[8]. What is central to *Mindstorms* is not "thinking" - it is rather "constructing", by computational means, concrete versions of abstract mathematical concepts; or, it is building personal mental models to understand the world - computational "environments" ("metaphors", "ideas", etc.) are one of the most effective and economic ways to obtain such models in an autonomous manner. The appeal of the computer is that *it provides a concrete reference for the abstract concepts* to be understood.

A naive reading of *Mindstorms* may give the impression that it backs the idea of the transfer of (meta-)skills from CT to other disciplines.

This seems even explicitly confirmed by Minsky [1970], where he emphasizes his shared view with Papert: "*our conjecture* [is] *that the ideas of procedures and debugging will turn out to be unique in their transferability.*"

On this count, however, it is useful to read Feurzeig et al. [1970][9], written in the same year of Minsky's lecture, where Papert and colleagues made claims on how "appropriate teaching with a suitable programming language can contribute to mathematics education". Let us review some of them, seeing how the initial naive idea of "automatic transfer from learning programming to learning math" is specified in a more precise and realistic idea of "using programming as a tool for experimenting and learning with math".

[7] However, we will discuss the work of Feurzeig et al. [1970] in details in the next section.

[8] The adjective "computational" is used 39 times in the book (CT is used only once). Among the expressions used more than once throughout the book we find: c. ideas (p. 17, 121, 145, 155); c. culture (p. 5, 100, 170, 174); c. metaphor (p. 105, 154, 169, 171, 187); c. model (p. 106, 164, 169); c. environment (p. 182 twice, 212);

[9] [Feurzeig et al., 1970] was written ten years before [Papert, 1980]. It reports on the first fifteen months of using the LOGO programming language in teaching mathematics to three classes: second, third and seventh grade.

The first claim seems indeed to support the idea of CT transferring to general skills, as already noted by Tedre and Denning [2016]: "*programming facilitates the acquisition of rigorous thinking and expression.*" However, this concept is explained further in the article: the peculiarities of computer programming make it a privileged tool for learning problem solving with an experimental approach[10]. In fact, children have to impose on themselves rigor and precision in instructing the computer - being explicit and precise is not imposed (incomprehensibly) by an enforcement of the teacher, but naturally emerges from the need of being understood by an automated executor with a limited instruction set, which is unable to perform any "human" inference. Briefly: the computer creates an intrinsic motivation to learn by trial, error and debug.

A next claim is that, again, "*programming provides highly motivated models*" for the so called *heuristic concepts* (e.g. "formulate plan", "separate the difficulties", "find a related problem", "contrast between global planning and formal details of a solution", "sub-goals and sub-problems", "debugging as a definite, constructive, plannable activity", and so on). Note again the emphasis on the fact that programming provides high motivations for learning these concepts. Moreover, note also that many of these ideas are included in modern CT definitions, often as CT "practices" or "approaches" (see Section 2.2). Bender [2017] even states that "heuristics" is the name given by Papert to what today we call CT.

Finally, Feurzeig et al. [1970] confirm that the purpose of their experiment is to use programming as a foundation for teaching mathematics, rather than teaching programming as a topic on its own (however recognising the importance of this second aim).

In summary, it is not the programming skills which count, or the "algorithmizing" concepts acquired through programming. A reading of *Mindstorms* which takes into account the entire book, and not single, isolated quotations makes clear that the focus is on the use of computers as formidable tools for "*addressing what Piaget and many others see as the obstacle which is overcome in the passage from child to adult thinking.*" "*Knowledge that was accessible only through formal processes can now be approached concretely. And the real magic comes from the fact that this knowledge includes those elements one needs to become a formal thinker*" [Papert, 1980]. Any rendering of Papert's position as a mere transfer of meta-skills would thus be a gross misunderstanding. The outcome that Papert and his group envisage is not the result of a generic exposure to computational concepts and education. Papert [2000] explains: "*In Mindstorms I made the claim that* [...] *the ability to program would allow a student to learn and use powerful forms of* [...] *ideas. It did not occur to me that anyone could possibly take my statement to mean that learning to program would in itself have consequences for how children learn and think.* [...] *Papers were written on 'the effects of programming (or of Logo or of the computer)' as if we were talking about the effects of a medical treatment.*" The modality of interaction with the computational media is as (and probably *more*) important than its contents. It is now high time to come back to the quotation about CT, and to read it in its context:

> I have no doubt that in the next few years we shall see the formation of some computational environments that deserve to be called "samba schools for

[10] Authors recognize that is theoretically possible to teach programming as an abstract mathematical concepts, without using computers, but this will cause the loss of the *essential* aspect of hands-on learning.

> computation." There have already been attempts in this direction [. . . , but] they have failed to make it because they were too primitive. [. . .] Their visions of how to integrate computational thinking into everyday life was insufficiently developed. But there will be more tries, and more and more. And eventually, somewhere, all the pieces will come together and it will "catch." [. . .] They will be manifestations of a social movement of people interested in personal computation, interested in their own children, and interested in education. [Papert, 1980, p. 182].

This reference to Brazilian "samba schools" should not be surprising, if we recall (Section 3.2) that Papert developed the *constructionist* learning theory. Very briefly, this theory takes from constructivism the idea of active building of knowledge through experience; it adds that learning is especially effective when learners are involved in the active construction of objects meaningful to them.

Papert was particularly critical towards traditional school systems. According to him, computers and programming will make old schools obsolete and useless, because learning will happen in constructionist environments, which would resemble traditional Brazilian samba schools, so fundamental for the preparation of the Rio Carnival. They are not schools in the traditional western meaning; they are rather clubs ranging from hundreds to thousands of people, from children to their grandparents, from novices to professionals. Members of each school gather every weekend to dance and to meet with friends. All of them dance: the novice learns, the expert teaches, but also practices for harder moves. There is a great social cohesion, a great sense of belonging, a strong idea of having a "common purpose." Although learning is spontaneous and natural, it is also *deliberate* - results of a year of work are spectacular, professional level representations, with references to traditions and with strong political undertones. All of this is present in Papert's reference to "samba schools of computations": environments where children and grown-ups may learn (by doing) the principles of computation, and use them to learn other disciplines, in a computational perspective. Their learning method will be radically different from what is common in traditional schools. No knowledge is transmitted, and pupils will learn because are immersed in an environment whose activities are both rich of computational principles and meaningful for the community.

That Papert's prediction about the revolution in the school system did not materialize, and his ideas in Mindstorms have been misunderstood and oversimplified. In retrospect, [Papert, 2000] reminds about the subtitle of the book ("Children, Computers, and Powerful Ideas"), acknowledging that both enthusiasts and detractors focused on the first two elements, forgetting the third, the most important one. Children are able to learn powerful ideas about the world (e.g., mathematical concepts like the idea of "zero", or "probabilistic thinking"), but these ideas have been disempowered by schools, which teach mathematics only through application of formulas, or by proposing problems situated in "fake" contexts that fail to be meaningful for pupils. The real thesis of Papert's CT, is not that "learning to program will *in itself* have consequences on how children learn and think", but that ability to program a computer can help re-empowering pupils[11], and bring to them powerful ideas about

[11] This is also made clear in the "Introduction to the second edition" of Mindstorms (1993), which critiques

mathematics, physics, probability (which are the fields touched upon in Mindstorms). He was thinking about computational environments where students can experiment with randomly generated objects, or use randomness to automate a programmable robot encountering some obstacles, or "rediscover" the concept of zero when working with a speed variable in a simulation or a videogame. Finally, Papert [2000] is optimistic on the fact that the change that was hoped for schools in the 1980s will eventually happen because of the increasing dissonance between school and society, and the increasing availability of technologies and ideas needed for the change to happen. We will return on this in Section 6.4.

Papert used again the expression in some of his important writings, but again only *en passant*.

The expression "computational thinking and practice" is used in Papert and Harel [1991] as opposed to "computer literacy" and "computer aided instruction (CAI)" (see 3.2) in relationship with the feminist battle. Despite the innovation of CAI, it seems to support "*the abstract and impersonal detached kinds of knowing*" of traditional schools. By contrast, "*many women prefer working with more personal, less-detached knowledge and do so very successfully*", and it is argued that Papert's CT moves towards this direction.

The expression is used again by Papert [1993, p. 184], talking about the origin of computers for the need of calculating missiles trajectories: "*many top mathematicians were mobilized to the task, among them John von Neumann and Norbert Wiener, who became* [...] *leading pioneers in the emergence of computers and of computational thinking.*" However, what is more interesting is that, in the same chapter, Papert [1993, ch. 9] envisages a new subject (that he proposes to call "Cybernetics for children", but that shares many characteristics with what we are calling "Papert's CT"). In facts, he proposes a subject that [Papert, 1993, p. 181-182]:

- is the "*kernel of knowledge needed for a child to invent*" and to build entities that resemble or evoke real life technologies;
- uses that kernel as a starting point for connections with other areas;
- uses "*technology as a medium for representing behaviors that one can observe in oneself and other people*";
- fosters a more intimate and affective relationship between the student and his work;
- has a more pluralistic underlying epistemology.

The expression was used again by Papert [1996b]. He was comparing two solutions to a geometrical problem: one used a powerful geometry tool to solve the problem as it would have been solved in a traditional pen and paper setting using Euclidean geometry, the other one used a Monte Carlo simulation. Both of these methods, says Papert, use a computational tool to effectively find a solution, but none of them make more clear the

---

the critiques of the first edition based on a supposed claim that "*'doing Logo' or 'working with computers' would* cause *change in how children think.* [...] *Logo does not itself produce good learning any more than paint produces good art.*"

underlying mathematics. So, "*The goal is to use computational thinking to forge ideas that are at least as 'explicative' as the Euclid-like constructions (and hopefully more so) but more accessible and more powerful.*" Papert then proposes a method to use Turtle geometry to deeply understand the mathematical concepts underlying the solution.

The expression "computational thinking" will return many times in Papert's last[12] talk [Papert, 2006], a few months after the publications of Wing's paper. The starting point is that the school system is dominated by graphocentrism, because it uses obsolete technologies - pencil and paper[13]. This reduces knowledge "*to the kind of knowledge that can be written down: propositions*" - a "propositional thinking" which is good for testing and grading students. LOGO was the first step beyond this paradigm, introducing "procedural thinking": knowledge as instructions, expressed through a programming language. But Papert acknowledges that this is only a first step towards *computational thinking*. One of the main aspect of CT is what he calls "object oriented thinking", not referring to the programming paradigm, but defining it as the "*making and understanding of computational objects*". These computational objects[14] may of course be used to teach programming, or standard geometry, but their intended role is another: they are objects you can "*get to know* [...] *more like the way you get to know a person*". Again: these are objects to think with, in a cognitive *and* in an affective sense. The great contribution of CT should be making "key, big ideas" of mathematics accessible to children, thus allowing to "turn learning upside down": in history, people started using and developing math for concrete aims, and doing so they "*developed something called* mathematical thinking", and only gradually this became the field of formal, pure mathematics. But nowadays in school this process is reversed: we wait to teach mathematical big ideas when pupils are ready to learn the abstractions needed to manage the formal part. With computers, they can start from the applications and gradually go up to abstractions.

6.4 The Digital Discontinuity

Despite rapid society evolution, school curricula, in any country, has significant inertia - it changes very slowly, and radical reforms are rare. They happen only when a significant fracture between the curriculum itself and the society it is supposed to represent occurs. This is what started to happen at the beginning of the twenty-first century: the digitalization of everyone's life, the substitution of information for the capital as the driving force of industrial innovation, the contraction of the perceived distances due to the availability of direct sources of information, started to become so prominent - and evident to everybody - that a request for school to cope with innovation became inevitable. In Fadel et al. [2015] this is identified as "the perfect learning storm", caused by four converging forces: knowledge work ("*steady supply of well-trained workers, using brainpower and digital tools to apply well-honed knowledge skills to their daily work*"); thinking tools (the necessity to use - and not be

[12] The very next day, Papert was struck by a motorbike and received a serious brain damage.

[13] The evocative image Papert proposes is that of an alien anthropologist visiting Earth and understanding that all knowledge workers adopted computers as their main work tool - all except students.

[14] The implicit reference is, of course, to LOGO's turtles.

overwhelmed by - the digital tools for thinking, learning, communicating, collaborating, and working); digital lifestyles (the naturalness of use of digital, mobile, ubiquitous tools requires that also learning become interactive, personalized, collaborative, creative, and innovative); learning research (developments in learning technology allow "*to personalize learning to meet each student's learning abilities and disabilities, learning styles and preferences, and unique profile of talents and competence.*") Moreover, Pierre Bourdieu's notion of "cultural capital" [Bourdieu, 1977] applies easily here: if school wants to maintain its role as a driver of social mobility, it has to change its approach, so that the digital resources that every student nowadays has, could become a capital for everybody, and not only for those children and young people who come from homes where that culture is present and that capital is already exploited (see also Merchant [2007]).

It is in this context that Wing's peroration [Wing, 2006] (see Chapter 2) about CT made its triumphant march, twenty-six years after *Mindstorms*, and after the dramatic rise of the digital society and computational sciences. On one side, we see the availability of digital tools to everybody, through the World Wide Web as the single infrastructure through which all different technologies are deployed. On the scientific side, computational tools are no longer the product of only computer scientists - most scientific disciplines become "computational", stably adding *simulation* as the third component of science, after theory and experiment [Winsberg, 2010]. Wing's paper was then published at the right moment, for selling CT to a broader audience than computer enthusiasts. From simple "algorithmizing", in Wing's hands, CT becomes a large umbrella for thought processes and techniques covering also natural information-processing.

The need to respond to the societal changes was matched by an educational offer, which was broad (and undefined) enough to appeal both at those wanting a formal presence of computer science in the curriculum, and at those who would instead go for mere "digital literacy", or simply "coding." The fact that CT was not operationally (or epistemologically) clearly defined may have even helped in its diffusion.

As Tedre and Denning [2016] says, "*Wing's formulation struck a resonant cord,*" and around that manifesto several interests coalesced into a movement to bring CT into all levels of K-12 education. Adding to the initiatives described in Chapter 1, let us mention an influential working paper of the OECD [Ananiadou and Claro, 2009], which led the way to the inclusion of CT in the PISA 2021 study[15], as part of mathematics.

6.5 Two Views, Two Misinterpretations, Two Fundamental Features for CSEd

Wing's CT lays in the path well-marked by the early computer scientists (see Section 6.2), extracting from computer science a list of thinking patterns ("*mental tools that reflect the breadth of the field of computer science*"), which include efficiency, approximation,

[15]The Programme for International Student Assessment (PISA) is a triennial international worldwide survey by the Organisation for Economic Co-operation and Development (OECD) for the evaluation of educational systems through the measure of the performance on mathematics, science, and reading of students in their 15th year. Mathematics is the primary domain assessed in the edition 2021, as it was in 2003 and 2012.

recursion, using abstraction and decomposition in computational problem solving, or exploiting "*reduction, embedding, transformation, or simulation*" to reformulate a difficult problem. In particular, Wing [2006, p. 35] is clear in stating that "*computer science is not computer programming. Thinking like a computer scientist means more than being able to program a computer*".

The inclusion of CT in the PISA 2021 study may be seen as the ultimate success of "Wing's computational thinking" - it made it into one of the longest-running and accepted international tests of the performance of students across disciplines. It may be read, however, also as a normalizing move of the establishment towards "Papert's computational thinking": instead of "turning learning upside down", the computational objects are integrated into the standard practice of traditional education, thus neutralizing their revolutionary potential.

We believe that both "Wing's CT" and "Papert's CT," two related but different views, have been misinterpreted. Paradoxically, this misinterpretation made them initially popular, but risks being their condemnation if their most profound - and most important - features are not considered and understood.

1. Wing's CT has been a way to shine a light on the urgent need to teach "core CS concepts", as a tool to understand and being an active participant in a digital society where computation is ubiquitous. Non computer scientists focused on the (unverified) claims that this computational problem-solving strategy - often simplified with "coding" - could easily transfer to every human activity. Teaching computer science should instead focus on the "big ideas", fundamental to understand and act in a digital world, rather than on the technical details. Since general ideas are hard to teach and transfer (see 4.2), details are of course needed, but only if instrumental for conveying the deep concepts, and always in the context of meaningful and deliberate learning.

2. Papert's CT has been a way to shine a light on the fact that technology was (and often, still, is) used to replicate traditional transmissive instruction (*instructionist Computer-Aided Instruction*). People focused on the unverified claims (only apparently supported by Papert) that learning programming could automatically result in better thinking skills. What should be kept in mind is, instead, that programming a computer can be a formidable tool (or, more evocatively, a meta-tool) for thinking and for more meaningful learning (both cognitively and affectively - like what happens in samba schools) by creating and experimenting.

The following subsections expand our thesis on these two crucial points.

6.5.1 Wing's CT[16]

To computer scientists, it appears clear the inseparable bond between CS and CT, expressed by Jeannette Wing [2006] in her seminal article.

[16]This section draws on Lodi, Martini, and Nardelli [2017], born as a translated and widely expanded version of a draft later published by Nardelli [2019].

However, to the general public, not explicitly trained in CS, these aspects may appear obscure or not so related to CS itself. That is why many educators, for example, tend to stick with the most understandable examples (but for which there is no scientific evidence [Guzdial, 2015]): those stating that CT can transfer to everyday life situations like preparing your backpack.

We also see many, well-published initiatives where CT is identified (more or less explicitly) with "coding" (that is, the last phase of the programming process), an equation that "*keeps spreading into the jargon of CS educational research, of CS curricular development (at all levels), of stakeholders such as politicians who determine or affect educational policies, school principals, and school advisors, among others*" [Armoni, 2016]. That computer science (and hence CT) is much broader than "coding" (indeed much broader than "programming") is something that computer scientists and educators have known for decades [Tedre, 2014] - accepting now the identification would be a significant step back. We must resist to the illusion that "coding" (as representative of any simplistic approach to the acquisition of the basic of computer science) could be a shortcut to the acquisition of that "algorithmizing" that early computer scientists viewed as one of the main contributions of their discipline to general culture.

6.5.1.1 CT Is Not a (New) Discipline, Cs Is

According to Armoni [2016], the problems stem especially when, instead of understanding CT as a cover for the scientific core of computer science, it is viewed instead as a new discipline, which can be taught and evaluated *as such*.

Important reforms seem to acknowledge that CS is the subject to be taught.

In England, the national computing curriculum[17], published by the Ministry of Education in September 2013 and mandatory for all schools starting from the school year 2014-15, uses the term CT only a couple of times. The term appears in the first sentence: "*A high-quality computing education equips pupils to use computational thinking and creativity to understand and change the world*", and then in objectives for Key Stage 3 ("*understand several key algorithms that reflect computational thinking [for example, ones for sorting and searching];*") and 4 ("*develop and apply their analytic, problem-solving, design, and computational thinking skills*"). The curriculum never defines the term.

In the United States, the mentioned "*Every Student Succeeds Act*" (ESSA), introduced computer science among the "*well rounded educational subjects*," and does not mention CT. Similarly, the *Computer Science For All* initiative was launched "*to empower all American students from kindergarten through high school to learn computer science and be equipped with the CT skills they need*".

In France, the Committee on Science Education [2013] recommends to teach Informatics: "*teaching should start at the primary level, through exposure to the notions of computer science and algorithms,* [. . . and] *should be further developed in middle and secondary school*". They use CT ("pensée informatique"), to talk about the specific habits of thinking developed

[17] https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study

by learning Informatics: "*computing* [...] *leads to a different way of thinking, called CT*" and "*learning about programming is a way to discover the rudiments of CT.*"

It is clear from these examples that CT is not considered to be a new subject, but as the *the conceptual sediment of disciplinary learning of CS*.

On the other hand, we must not let the criticisms of the idea of computational thinking (e.g., the fact that some of its characteristics are shared with other disciplines - see 2.2.1) diminish the value of CS as an autonomous discipline.

Recalling the Cuny-Snyder-Wing definition (see 2.2), it is essential to stress the role of the **information processing agent**. Without the agent and its ability to operate **effectively**[18], there is no computer science, only mathematics, which, in fact, has solved problems for millennia, discovering and applying, for example, abstraction, decomposition, and recursion in this long journey. Of the same opinion is Alfred Aho [2011], who, immediately before stating the definition that inspired Cuny, Snyder, and Wing - underlines how the term "computation" should be used in conjunction with the indication of a well-defined computational model: since computation is a process defined in terms of an underlying model, this cannot be ignored.

This brings a radical viewpoint change with respect, for example, to mathematics. The conceptual paradigm shift is the transition from "solving problems" to "make something solving problems for us." Unlike other disciplines, *CS abstractions can be executed mechanically* [Wing, 2008]. This means that CS abstractions can be "animated" without having to build a new physical representation of abstraction itself. Moreover, CS, through its "languages," provides linguistic support for abstraction: appropriate linguistic constructs allow us to define abstractions in a systematic way, and to move uniformly between their various levels arriving "down" to the physical execution of the identified abstractions. In this, computer science is distinguished from other disciplines, which must build each time "ad hoc" material objects that express the phenomena modeled. This ability to build "executable abstractions" makes CS an important aid for the study of other disciplines as well.

6.5.1.2 Why to Teach CS

Some detractors (see, for instance, Mannila et al. [2014] for a balanced review) of CT accuse computer scientists of being arrogant in wanting to "teach everyone how to think" or to want to transform every child in a software developer.

However, as already discussed in Chapter 1, the aim is to teach CS to give tools to understand and act in our digital world.

Linked to that, already in Wings' original paper, and then in subsequent works, computational thinking has been referred to as the "fourth basics" to be taught in addition to the classic "reading, writing, and arithmetic"[19]. This idea also emerges in older works: for example, in the words of Forsythe mentioned at the beginning. It has to be noted that the 2012 "Italian National guidelines for the curriculum" for primary and lower secondary[20] underline that the task of the first cycle of education is to promote "basic cultural and social

[18] Let us remember that one procedure is **effective** if it consists of a sequence of elementary actions, executable in a deterministic way and in a finite time by the processing agent.

[19] The "fourth R", together with "Reading, wRiting, aRithmetic".

[20] http://www.indicazioninazionali.it/2018/08/26/indicazioni-2012/

literacy" to acquire languages and codes of our culture (extended to other cohabiting cultures and awareness in use of new media). It is specified that this literacy *includes* instrumental literacy ("reading, writing, and arithmetic"), and enhances it through the languages and knowledge of the disciplines. This, therefore, frames the "three basic skills" - whose learning was the goal of literacy over the centuries - as *tools* for a higher task: learning to understand the social context in which the student finds himself living. Tools that, in today's complex world, need specific languages and knowledge from different disciplines in order to provide adequate basic training. CS, we believe, should be one of these disciplines. Of course, since transfer is not automatic (Chapter 4), it is important to explicitly link CS concepts that we teach to the "computational world" that the students will found outside school.

6.5.2 Papert's CT

Papert's ideas are often framed like "Programming can teach students to think logically," similarly to what has been said for decades for the teaching of Math, Latin, or ancient Greek (see Chapter 4). As seen, this may be true, but not because an *automatic competence transfer* happens (research shows it is very difficult), but because "engaging in intellectually demanding tasks is important for student's cognitive development" [Lewis, 2017]. As already described in Section 4.4, during the 80s, after Papert's proposals, some research about the cognitive effects of programming has been conducted, leading to mixed results [Scherer, 2016]. Papert himself acknowledges that what he calls the "Latinesque" justification for teaching something is not sufficient: one should always test if there are other ways to achieve the same goals [Papert, 2006]. It should be clear from the previous sections, however, that he thought computers and programming have some peculiar characteristics that make them particularly useful for the training in logical thinking, but this happens only if they are used as constructionist tools: to create meaningful artifacts. Papert [1996a] itself acknowledges that transfer it is something that does not happen automatically, and needs an active and deliberate effort:

> [...] the problem of *transfer*: If you learn something in one context, can you use it in another? [...] The answer is simple: depends on what you actually learned. Of course the skill won't transfer if what you learned was a meaningless ritual [...]. But if you understood the principles, and if you have an attitude of self-confidence in trying and modifying, your old experience will let you find out quickly what to do in the new situation. *Transfer is not something that happens to you. It's something you do.*
>
> [Papert, 1996a, p. 125-126, emphasis as in original]

In this framing, we also have to be cautious regards the claim "*programming helps students learn Science and Math*". While, again, it is unlikely that learning to program will directly transfer in better learning of other STEM disciplines [Lewis, 2017], studies from the 80s (for a review, see [Guzdial, 2015, pp. 41-49]) show that students can learn better some scientific concepts with the help of *specifically designed programming environments* (LOGO being the originating one). By contrast, some (but not all) of these studies found

that students learned only those basic concepts of CS/Programming that were needed to interact with the learning environments.

Much more interesting, and in line with Papert's CT, is the idea that *programming provides emotional value, agency, and motivation*. Indeed, in the constructionist spirit, programming can be "a medium for creation, communication and creative expression" [Lewis, 2017]. Kafai and Burke [2014] show examples of students using programming and "making" to create, and then to connect with others while sharing their creations. Of course, programming is not the only medium useful for this; however, the fact that through programming we can simulate (and make concrete) many physical or abstract processes, gives it a particular educational role. Moreover, the availability of mobile devices and the ease of connection and sharing through the Internet make programming a privileged, central tool for such kind of "samba schools."

Likewise, for Wing's CT, we should care less about the (improbable) automatic transfer between learning CT and other skills.

From Papert's CT, we must preserve the innovation potential of the samba schools of computation, without let them being marginalized into after school educational initiatives. We are now aware of the huge potential that comes from making learning constructive and meaningful for students - it re-empowers powerful, big ideas (of mathematics, physics, and potentially any other discipline) that have been disempowered by the school. Indeed, solving problems using computers and computer science principles forces one to think logically/systematically/procedurally, not because of external imposition, but in virtue of the computers' intrinsic nature. Moreover, constructing computational artifacts gives the opportunity to have computational objects to identify with, and to "concretely" manipulate them, to explore and build with them. Furthermore, this has to happen in all "standard" schools, because it is there that it is necessary, where the transformation of digital resources into a cultural capital is an imperative for social inclusion.

If every subject (CS included) is taught in a more constructive, meaningful, and creative way (see 3.3), like in a "samba school," this will contribute to the empowering of all students, in particular the young and the disadvantaged.

6.6 Conclusions

The expression "computational thinking" is an abused buzzword in K-12 education, which means different things to different people, especially in its relation to Computer Science, and in its relevance to the learning of other disciplines.

At least two leading researchers have used the expression: Seymour Papert (in 1980) and Jeannette Wing (in 2006), each of them stressing (when more, when less explicitly) different aspects of a common theme.

We conducted historical research, and literature review, with retrieval and discussion of the original sources. For Papert, we discussed the occurrences of the expression in his 1980's Mindstorm, and in subsequent papers, some of them explicitly addressing the misunderstanding of his idea. Previous Papert's papers are also analyzed, because they are relevant to link this research to the crucial idea of transfer of competences.

"Wing's CT" and "Papert's CT," when correctly understood, are both relevant to today's computer science education. Both of them suggest (or seem to suggest) that the competencies acquired as CT will transfer to other disciplines. This largely unverified (when not disputed) claim has also been a reason for their appeal.

However, it is not the possibility of transfer that matters for the role of CT in K-12 education. What matters is maintaining both the two different dimensions proposed by Wing and Papert in current CT. From Wing, we should retain the centrality of computer science, CT being the (scientific and cultural) substratum of the technical competences. In this way, CT becomes a lens and a set of categories for understanding the algorithmic fabric of today's world. From Papert, we should retain the constructionist idea that only a social and affective involvement of the student into the technical content will make programming an interdisciplinary tool for learning (also) other disciplines, in a more constructive, meaningful, and creative way, and a way for re-empowering disciplinary big ideas.

Chapter 7

A Curriculum Proposal[1]

In light of the motivations described in Chapter 1, we recognize the importance and urgency to bring Computer Science, its ideas and ways of thinking to all Italian primary and secondary school students, most of which are not taught this subject as part of an official curricular requirement.

A working group (in which the author of this thesis participates) of the Italian Inter-universities Consortium for Informatics (CINI), in collaboration with the academic associations who gather together researchers in informatics (GRIN) and computer engineering (GII), has recently proposed a core informatics curriculum for all the levels of compulsory school.

This chapter summarizes the proposed curriculum, highlights the key underlying motivations, and outlines a possible strategy to ensure that its implementation in schools can be effective.

The full English version of the proposed curriculum is reported in Appendix B.

7.1 Context, Process and Background of the Proposal

In this chapter we present a recent proposal [Nardelli et al., 2017] on behalf of our Italian academic community, which is meant to contribute to the development of informatics education in the Italian primary and secondary school. It is the outcome of a long process, which has also benefited from important contributions of pedagogists and experienced school teachers who took part in the discussion. We are nevertheless aware that it is just a step in a longer path to make all people in charge of school policies aware of what is at stake for the students' future.

7.1.1 Writing and Revision Process

The proposal of a core informatics curriculum for all the levels of compulsory education is a recent initiative of the Italian Inter-universities Consortium for Informatics (CINI), promoted by its interest group for "Informatics and School", and carried out in cooperation with the

[1] This chapter is based on material published by Forlizzi, Lodi, Lonati, Mirolo, Monga, Montresor, Morpurgo, and Nardelli [2018].

academic associations who gather together researchers in informatics (GRIN) and computer engineering (GII). The process started in June 2017, when the assembly of the interest group charged an editorial board to write a first draft of the proposal. The draft was subjected to discussion and refinement during several distance meetings of the editorial board[2], held between June and early August. The document was then made available to all members of the CINI interest group for Informatics and School, who contributed some valuable feedback and were eventually able to reach an agreement on it in a meeting convened in September 2017. The next step was to publish the proposal in the form agreed upon by the interest group and to invite the whole informatics community (represented by CINI, GRIN and GII) to provide comments and suggestions. In the meanwhile, further contributions were collected through discussions with experienced teachers, pedagogists and other experts of school policies. By the end of October 2017 the editorial board examined the feedback from the informatics community and edited the current revision of the document. The revised proposal obtained an official status in November, after formal approval from the assemblies of CINI, GRIN, GII. Finally, in December 2017 it was presented within an initiative of the Italian Chamber of Deputies.

For convenience, the proposal (Appendix B) has been organized in conformity with the competence-based model of MIUR documents reporting curricular recommendations. It attempts to introduce the general educational motivations as well as to explain as clearly as possible our community's cultural and scientific perspective. The discussion is currently continuing on practical issues, such as the need for instructional material and the involvement of in-service teachers. In March 2018 the document was also submitted to the chair of the MIUR Scientific Committee for the National Curricula for the primary school.

7.1.2 Other Curriculum in Europe

Our proposal draws on three noteworthy curricular models:

1. The 2011 version of CSTA/ACM K-12 standards [Seehorn et al., 2011], which identify five major strands: *computational thinking*, *collaboration*, *computing practice and programming*, *computers and communication devices*, and *community, global and ethical impacts*.

2. The English implementation of the new computing subject [Department for Education, 2013], structured into three components emanating from the Royal Society report [The Royal Society, 2012]: *computer science*, *information technology,* and *digital literacy*.

3. The report of the French Academy of Science [Committee on Science Education, 2013], which distinguishes between three ways of learning informatics, each appropriate for a different instruction level: *discovery* (primary school), *acquisition and autonomy* (lower secondary), and *mastering concepts* (high school).

[2]Composed by: Enrico Nardelli (coordinator), Luca Forlizzi, Michael Lodi, Violetta Lonati, Claudio Mirolo, Mattia Monga, Alberto Montresor, Anna Morpurgo.

The five key areas that have been identified to organize our informatics curriculum - *algorithms*, *programming*, *data and information*, *digital creativity* and *digital awareness* (see next section) - are closely connected with the aims listed in the computing curricula for England [Department for Education, 2013].

7.2 A Core Informatics Curriculum

The proposed curriculum has been conceived in a two-dimensional framework. The former dimension, starting from grade 1 of primary school, is characterized by three main learning stages:

1. In the first stage (primary school) pupils are encouraged to *ask questions*, as well as to *discover* in their everyday life and to *explore* some basic ideas of informatics. They can be engaged either in *plugged*, i.e. implying the use of computing devices, or *unplugged* activities, i.e. without using digital technologies (see Section 8.4).

2. In the second stage (lower secondary school) students are expected to *grow in autonomy*. To achieve this educational objective, they have to learn more about the organization of data and the concept of algorithm. Moreover, they should be offered opportunities to develop *abstract thinking* and to acquire new specific as well as *cross-disciplinary* skills. In particular, programming tasks can play a key role in this respect.

3. The first two stages lay the foundations for mastering the concepts and for enhancing the skills at the core of the third stage (upper secondary school), at the end of which students should be able to *model problems* and to *design algorithms*. Abstraction, organization and accuracy are essential traits of the problem solving approach in the informatics field, and can foster the development of critical thinking and provide helpful keys to master complexity.

The latter dimension concerns the content, which is organized into five key areas: *algorithms*, *programming*, *data and information*, *digital creativity* and *digital awareness*, described in the following subsections.

Overall, as reported in the list of general learning goals, at the end of compulsory school each student should be able:

- to understand and to apply basic concepts and principles of informatics;
- to approach problems by exploiting tools and methods of the field;
- to solve problems by devising formal representations, by designing algorithms and by coding the algorithms in a programming language;
- to evaluate the potential benefits as well as the limits of applying a range of digital technologies to achieve a given task;
- to use digital technologies in a conscious, responsible, confident, purposeful, active and creative way.

7.2.1 Area of Algorithms

Algorithms are at the core of informatics. They predate programming, as several noteworthy algorithms have been designed well before the advent of computers. Pupils should meet the concept of algorithm since the early years of primary school, in an informal and playful way at first. Starting from the lower secondary school, the level of formality is progressively increased and the concept of algorithm is linked to other school subjects. By the end of the curriculum, pupils are expected to master the notion of algorithm and the related scientific concepts. To achieve these general goals, the curriculum tackles four main topics.

Algorithms as procedures. Pupils first encounter algorithms in grades 1-3, as a way to describe the procedures representing the activities of everyday life; e.g., brushing one's teeth, dressing, leaving the classroom in an emergency drill. The initial approach could be unplugged, to later evolve into plugged activities, for example to solve coding puzzles. In grades 6–8, the collection of processes that are studied algorithmically is extended to include examples taken from other disciplines, such as mathematics, science and technology. Towards the end of the curriculum, pupils should know a selection of simple algorithms that solve fundamental informatics problems such as search and sorting. Apart from studying existing ones, pupils should progressively grow in autonomy and start to design their own algorithms, a skill that should be accomplished by the end of the curriculum.

Interpretation and disambiguation. The process of learning and designing algorithms should be accompanied by an increasing understanding that algorithms need to be described in a precise and unambiguous manner. In grades 1–5, this goal could be accomplished by having the pupils perform the role of the executor, in an unplugged way. In grades 6–8, the need of precision is reinforced, by making pupils reflect on the instructions performed by the automatic executor and how they are always completed in the same way. Pupils are thus expected to reflect on the ambiguities hidden in an algorithm described using natural language.

Decomposition. By grade 3, pupils should understand that difficult problems could be solved by breaking them down in smaller parts; by grade 5, such understanding should become operational, i.e., pupils should be able to actually solve simple problems in such way. These concepts are later reinforced by the concept of modularity introduced in the area of programming (see 7.2.2).

Reasoning about algorithms. During the second and third stage of the curriculum (lower and upper secondary school), pupils are introduced to a larger spectrum of issues related to algorithmics. They should move away from the concept of algorithms solving specific instances of a problem, and understand that algorithms should solve problems in their generality. By grade 8, pupils should be able to reflect on the correctness of their solutions, in particular by detecting and describing the conditions under which these processes can terminate. By grade 10, pupils should also be able to evaluate, in simple terms, the efficiency of basic

algorithms and use logical reasoning to evaluate different algorithms that solve the same problem. They should also be able to understand that not all problems can be solved by algorithms in an efficient way.

7.2.2 Area of Programming

Starting in the early grades pupils should get familiar with writing computer programs. In primary schools pupils write structurally simple programs, that possibly react to events, within a friendly - e.g., visual - programming environment. By grade 8, pupils are expected to design, write and debug, using easy-to-use programming languages, programs that apply selection, loops, variables and elementary forms of input and output. By grade 10, pupils are expected to comply with syntax while writing simple programs in a textual programming language; moreover they should be able to define, implement and validate programs and systems that model or simulate simple physical systems or familiar processes, that occur in the real world or are studied in other disciplines. Overall, pupils should be able to operate on a program in order to understand its behaviour, modify it, identify and fix flaws. In primary schools, pupils first observe errors in programs and act spontaneously to correct them, then they start examining programs in order to detect and fix errors and should be able to use logical reasoning to understand why a simple program fails; by grade 8 they should intentionally experience small changes in a program to understand and modify its behaviour; by grade 10 they should recognize how the various parts of a program contribute to its functioning, and be able to predict the outcome of a program without running it. To achieve these general goals, the following specific skills should be progressively developed from grade 1 to 10.

Sequencing, selection and iteration. In early primary school pupils are expected to sort a sequence of instructions correctly, use one-way selection to make decisions within simple programs, and explore the use of two-way selection to implement mutually exclusive actions. The first use of loops in primary school is to concisely express that a certain action has to be repeated a given number of times; then loops can be used to repeat a certain action while an easy-to-test condition holds. By grade 8 pupils should be able to nest selection and loops as above, and start using variables in the conditions of selections and loops; by grade 10, they should be able to write conditions that use a logical operator, and use conditionals/selections within loops to describe the repetition of parametric actions.

Use of variables. In primary school, variables are used to represent input and output data, or to represent data computed during the execution of a program; by grade 8, simply typed variables are used to represent the state of a program and track the progress of the computation; by grade 10, students should write programs with structured variables, and be able to use variables in loops to define exit conditions or parametric actions.

Modularity. By grade 5, pupils recognize that a sequence of instructions can be considered as a single action subject to repetition or selection; by grade 8 they should be able to

re-arrange a program to improve it, by organizing it in modular components as functions and procedures; by grade 10 they should design and develop modular programs using procedures and functions.

7.2.3 Area of Data and Information

The possibility of representing information through symbols, which can be stored and manipulated by an automatic processing system, lies at the very foundations of computing and then must be part of any informatics curriculum. Representing information is inherently connected to an abstraction process. Therefore concepts and methods in this area are acquired throughout the whole span of the curriculum, following the progression of pupils' abstraction abilities. Starting at grade 5, the curriculum aims at developing awareness that computers deal merely with raw data, encoded as symbols, and that information pertains only to the sphere of meaning, intrinsic to the human mind, what necessarily implies some degree of subjectivity. The main goals of the curriculum in this area can be classified by theme as follows.

Data Representation. Starting from grade 1, pupils gradually explore potential representations of various kinds of data (e.g., numbers, images, sounds), using different formats, possibly even some of their own conception. By the end of grade 3, they should be able to select and use suitable items to represent simple data they are familiar with (e.g., colors, words). At the end of grade 8, pupils should realize whether two alternative representations of the same data are interchangeable for a given purpose. The conventional character of any data representation, relative to what it is meant to describe, should be fully understood in grades 9–10. As a consequence, pupils become aware that different ways of representing data may affect both the effectiveness and the efficiency of a computation on such data. This achievement is also a prerequisite for the subsequent development of the ability to identify and choose the data representations best suited to an intended purpose.

Structure and organization of data. At grades 4–5 pupils start to represent simple structured data (e.g., bitmap images) as well as, through combinations of symbols, a little more complex data familiar to them (e.g., secondary colors, sentences). From grade 6 to 8 they develop the ability to classify data according to their kind (e.g., numerical, textual), that leads to the data type concept. At the same time, they should learn to perform simple manipulations of symbols that represent structured data (e.g., binary numbers, bitmap images), and to use structured variables to represent collections of homogeneous data (e.g., vectors, lists). At the end of grade 10, pupils should know the features of basic data structures (e.g., lists, vectors, matrices, dictionaries) and learn how to select an appropriate structure to approach a given problem.

Roles of data. The perception that data can be used in fundamentally dissimilar ways is to be developed in parallel with the programming skills, in particular those in connection with the use of variables. At the end of grade 8, pupils should be capable to distinguish the

different roles played by the data within a program. Starting with the identification of input and output data, pupils should become familiar with the representation of the state of a computation carried out by a computer program. The idea of metadata should be introduced, in some specific context such as HTML or a simple data description language, in grades 9–10.

7.2.4 Area of Digital Creativity

Often people think about the "big C" Creativity of famous artists or Nobel prices. But everyone can be creative: scientists developing new theories, doctors diagnosing a disease or entrepreneurs developing new products. And, in general, everyone coming up with a new idea in everyday life that's new, useful or interesting for herself is practising the "little c" creativity [Resnick, 2017a]. This is more and more crucial in a world where knowledge is easily available and boring stuff is easily automated, while rapid changes need constant innovation, adaptability and solution to ever-new problems (see Section 3.3).

Information Technologies (IT) are a very powerful means of self-expression and creativity. Starting from grade 1, pupils should become aware that they *can* use IT to express themselves, whereas too often they are just passive consumers of ready-to-use technological products and applications. As their ability to program improves, they are encouraged to engage with actively creating digital content and computer programs (in suitable environments), progressively using and combining different media, technologies and services. Moreover, they should start to reflect, to decide whether to use or not available technologies, and possibly to select appropriate technologies for different expressive purposes or to solve small problems they are personally interested in.

Use and creation of digital content. Using digital content and computer applications is just a first step: in grades 1–5, pupils should learn how to create simple and multimedia digital content; in addition, they start to select appropriate content, as well as to modify and combine it in simple ways. In grades 6–8 pupils should experiment with different ways of processing digital content (e.g., bitmaps versus vectorial images), while learning how to effectively present it.

Active creation of software applications. From grade 3, pupils should start to create simple computer applications like stories, games, music, using environments designed for their age (e.g., visual programming languages with blocks). In grades 6–8 they should be able to take advantage of their increasing experience with programming to create applications. In grades 8–10, pupils should use more advanced environments (e.g., text-based programming languages) to create more elaborate content. Moreover, they should combine programming and on-line services to solve problems and to achieve tasks.

7.2.5 Area of Digital Awareness

Computer-based devices have pervaded everybody's life and it is important to develop awareness in pupils, since early years at school, with respect to their use and how they affect life and relations. This goal is pursued by our curriculum along two paths: a first one focusing

on expanding pupils' knowledge of the information technology systems and devices, and a second one where students reflect, in increasing depth, on the personal and social impact of digital technologies.

Knowledge of Information Technology. In primary school pupils progress from recognizing the presence of IT devices all around them to being able to identify their main components and the main services they provide, while becoming aware of the importance of protecting personal data also in their various digital instantiations. In lower secondary school this knowledge is deepened with a comprehension of the main physical and functional principles at the core of computing systems and their communication networks, and through first experiences of interconnecting computer-based systems and input-output peripherals, and collecting and analyzing data. In the early upper secondary school pupils understand the importance of taking into account enduser requirements for the development of computer-based applications and deepen their experience with using computers to interact with and control external devices.

Social impact. In primary school pupils are progressively sensitized to the importance of interacting respectfully with others, even when using digital platforms, and to identifying and reporting problems in social interactions mediated by information technology. In lower secondary school they grow in their understanding of the value of data, both from a personal viewpoint and from a general one, and of how the collection and processing of large quantities of data affects society. In the early upper secondary school pupils arrive at critically reflecting on the multifaceted relations between information technology and society, spanning any domain of interest, and on the importance of keeping human beings in control of critical steps whenever computer-based decisions affect people.

7.3 Conclusions and Future Perspective

In order to have a real impact on schools we need to face several challenges and operational difficulties. A major challenge is to cope with the general shortage of teachers with sufficient familiarity with the basic concepts of informatics (its *Content Knowledge*), an issue emerging also in other countries [The Committee on European Computing Education (CECE), 2017; The Royal Society, 2017]. Even in vocational schools it is common to find teachers with a poor background in the field [Bellettini et al., 2015] and, given the current state of Italian the recruitment process, we can hardly hope that the situation will improve in the next few years. Thus, the teachers at all levels, but especially at the earliest grades, need support to cope with the need to reshape their teaching practice.

A national effort is then required to identify and validate suitable instructional methodologies and learning materials that can support effective learning by distilling and formalizing the *Pedagogical Content Knowledge* - i.e., the "*the knowledge of teachers to help others learn*", including "*the ways of representing and formulating the subject that makes it comprehensible to others*" [Shulman, 1986].

We believe this could be achieved by designing and carrying out small-scale action-research projects in cooperation between school teachers and researchers in computer science education, as well as by documenting the results, strengths and weaknesses of a variety of teaching approaches.

However, the next crucial steps to be successful in introducing informatics in the Italian school are to design a systematic way to disseminate best practices and to make available appropriate materials (e.g., textbooks, teaching resources) - often present but not translated in Italian.

The recommendations outlined here may still sound a little utopian to anyone who knows the current state of the Italian schools. In fact, our main goal was to raise awareness among stakeholders and, in particular, policy makers by offering a comprehensive view of how informatics could be introduced in schools as well as by pointing out the potential benefits for every citizen of the 21st century.

Our proposal has gained the endorsement of the reference informatics communities and is based on a sensible pedagogical analysis of its major themes. Moreover, as we have tried to explain in this chapter, the informatics curriculum has been conceived as a whole, built up of strongly interconnected parts: the learning goals of later grades would appear much more plausible in light of the achievements expected at earlier stages.

Finally, we are well aware that the strategy we envisage presupposes a strong political commitment towards informatics education in schools with a focus, beyond the hype of digital competences, on the scientific principles underlying the development of a digital society.

Chapter 8

Learning CT in a Constructive Way[1]

Currently, the most widespread methodology to teach computational thinking is teaching to program, often with languages and environments suitable for learner's age and experience (for example, for young children and beginners, environments where instruction are not textual code but visual elements that must be combined together to create a videogame or an animation). Another widespread methodology involves the so-called *unplugged activities* (the most being "CS Unplugged") where students are taught computer science concepts like algorithms, information encoding, cryptography, and so on through traditional games that do not need technology but material like pen and paper or simply student's own bodies. Particularly interesting for computational thinking are games where one child embodies the programmer and another one the programmed agent.

First of all, we review two classical concepts in computing education research: programming misconceptions and the struggle of students forming a viable "notional machine," to grasp the complex relationship between the program itself (the text of the code) and the actions that take place when the program is run by the interpreter.

After that, we review some historical languages for teaching, that paved the way for modern environments, which are as well described, with particular emphasis on their constructivist characteristics.

We then move to Unplugged activities, describing their constructivist (e.g., re-discovery of algorithms) and constructionist characteristics (e.g., kinesthetic activities to teach abstract concepts), and reviewing critical research, indicating that they can be a useful tool if used in combination with plugged one.

We finally discuss about the common elements between constructivist approaches (and in particular creative learning) with agile software development methods.

8.1 Constructionism and Learning to Program

Programming can be seen as naturally constructionist (see Section 3.2), in the sense that it always involves the production of an artifact that can be shown and shared. Of course, when

[1]This chapter is based on the material published in [Monga, Lodi, Malchiodi, Morpurgo, and Spieler, 2018; Lodi, Malchiodi, Monga, Morpurgo, and Spieler, 2019; Bell and Lodi, 2019a;b].

teaching programming, this aspect can be stressed or attenuated.

Failure rates and dropout percentages in traditional programming courses and the urge to introduce programming early in school curricula have fostered new approaches to teaching programming. In particular, the following points are given particular consideration:

- *motivation:* programming tasks should be engaging, to keep pupils' motivation high;
- *syntax:* novices should be introduced first to the logical aspects of programming and only at a later stage to the syntax;
- *a constructivist approach:* the construction of knowledge is to be fostered through activities that allow to discover important ideas, to work in groups and share meta-cognition;
- *a constructionist approach:* the production of meaningful personal projects and artifacts must be encouraged.

In this perspective, for educational purposes, visual programming languages (Section 8.3), have been developed and unplugged activities (Section 8.4) have been designed. In particular visual programming languages allow novices to concentrate on the logical aspects of programming without having to strive with unnatural textual syntactic rules. Moreover, they make it possible to realise small but meaningful projects, keeping students motivated, and support a constructionist approach where students are encouraged to develop and share their projects - video games, animated stories, or simulations of simple real world phenomena.

Approaches that exploit this characteristic have also been developed in software engineering (Section 8.5) and are being successfully applied to software development.

Before reviewing those approaches in a constructivist and constructionist light, let us review (Section 8.2) some important concepts that, in a cognitivist (see 3.1.2) or in a cognitive-constructivist (see 3.1.3) light, have been studied about the so called "Psychology of Programming".

8.2 What Does It Mean to Learn Programming?

The basic idea behind programming - i.e., producing a precise description of how to carry out a task or to solve a problem - is that an *interpreter*, different from the producer of the description, can "understand" it and effectively carry out the task as described. There are thus two distinct but closely related aspects in programming:

1. the program itself (the text or other streams of symbols or actions that build up the digital implementation of an algorithm),
2. the actions that take place when the program is run by the interpreter.

This distinction is explicit in most of the professional programming environments, but it is conceptually present even in those environments designed for very young children, where the

Figure 8.1: Bee-bot
CC0 Public Domain

program is somewhat implicit. The Bee-Bot[2] (see Figure 8.1), for example, is a bee-shaped robot that can be programmed by pushing the buttons on its back: the program, while recorded and then executed by the machine, is not explicit nor visible in its static form by the children, but it exists, and the programmer needs to master the relationship between the actions she records into the bee and the actions the bee will perform when the program will be executed.

In this chapter, however, we focus on programs in which the source code - either in the form of graphical blocks or in textual form - is explicit. Thus, one needs to have a mental idea of the *interpreter* in order to successfully program, in particular one needs to know:

- the set of basic actions the interpreter is able to perform,
- the language it is able to understand, with rules on how to compose basic actions,
- the relation between *syntax* and *semantics,* that is what actions it will perform given a description, and, conversely, how to describe a given sequence of actions so that it will perform them.

The first aspect, that is the program *source code,* is explicit, visible. The second one - the actions that take place when the program is run, is somewhat implicit, hidden in the execution time world, and not so immediate to grasp for novices. Moreover, this aspect is sometimes underestimated by both teachers and learners: disciplinary teachers, as experts, give it for granted; learners tend to construct personal, intuitive, but not necessarily coherent, ideas of what will happen. To help novice programmers take into account also the dynamic side

[2] https://www.terrapinlogo.com/products/robots/bee-bot/beebot.html

of programming, the concept of *notional machine* [Du Boulay, 1986; Sorva, 2013] has been proposed. A notional machine is a characterisation of the computer in its role as executor of programs in a particular language (or set of languages, or even a subset of a language) for didactic purposes. It thus gives a convenient description of the association syntax-semantics. The following learning outcomes should therefore be considered when teaching to program:

- the development by students of a perception of programming that does not reduce to the mere production of code, but includes relating instructions to what will happen when the program is executed, and eventually comes to include producing applications for use and seeing it as a way to solve problems;
- the development of a mental model of a notional machine that allows them to make the association (static) syntax - (dynamic) semantics and to trace program execution correctly and coherently.

In particular, this latter outcome goal will include the development of the following skills:

- given a program (typically one's own) and an observed behaviour:
  - identify when debugging is needed because the behaviour is somewhat not the one intended,
  - identify where a bug has occurred,
  - be able to correct the code;
- given a program and its specification, be able to test it;
- understand that there can be multiple correct ways to program a solution.

If these are crucial points in learning to write executable descriptions, however, programming is indeed a multifaceted competence, and the knowledge to construct and the skills to develop span over several dimensions, besides predicting concrete semantics of abstract descriptions. A skilled programmer needs to:

1. understand general properties of automatic interpreters able to manipulate digital information;
2. think about problems in a way suitable to automatic elaboration;
3. devise, analyze, compare solutions;
4. adapt solutions to emerging hurdles and needs;
5. integrate into teamwork and be able to elicit, organize, and share the abstract knowledge related to a software project.

Here we mainly focus on skill 1 and the support provided by programming languages and environments. Moreover we highlight the opportunity provided by agile methodologies to develop skill 5.

8.2.1 Notional Machines

Alan Perlis, in his foreword to "Structure and Interpretation of Computer Programs" (SICP) [Abelson et al., 1996], sets the stage for learning to program: "*Our traffic with the subject matter of this book involves us with three foci of phenomena: the human mind, collections of computer programs, and the computer. Every computer program is a model, hatched in the mind, of a real or mental process.*" And then: "*The source of the exhilaration associated with computer programming is the continual unfolding within the mind and on the computer of mechanisms expressed as programs and the explosion of perception they generate. If art interprets our dreams, the computer executes them in the guise of programs!*" .

This seems an important intuition for approaching programming from a constructivist perspective: programs are a join point between our mind and the computer, the interpreter of the formal description of what we have in mind. Thus, programs appeal to (or even exhilarate) our curiosity and ingenuity and are wonderful artifacts to share and discuss with other active minds. Such a sharing, however, assumes that the interpreter is a shared knowledge among peers. When a group of people program the same "machine", a shared *semantics* is in fact given, but unfortunately people, especially novices, do not necessarily write their programs for the formal interpreter they use, rather for the *notional machine* [Sorva, 2013; Berry and Kölling, 2014] they actually have in their minds.

As seen, a notional machine is an abstract computer responsible for executing programs of a particular kind [Sorva, 2013] and its grasping refers to all the general properties of the machine that one is learning to control [Du Boulay, 1986]. The purpose of a notional machine is to explain, to give intuitive meaning to the code a programmer writes. It normally encompasses an idealized version of the interpreter and other aspects of the development and run-time environment; moreover it should bring also a complementary intuition of what the notional machine cannot do, at least without specific directions of the programmer.

To introduce a notional machine to the students is often the initial role of the instructors. Ideally this should be somewhat incremental in complexity, but not all programming languages are suitable for incremental models: in fact most of the success for introductory courses of visual languages or LISP dialects (see next section) is that they allow shallow presentations of syntax, thus letting the learners focus on the more relevant parts of their notional machines[3].

The idea that it is fundamental for students to have an accurate understanding of the underlying machine can be recognized in Knuth [1997]. He uses, in his classic book "The Art

[3]On the other hand, several languages of widespread use by experienced programmers are in this sense more complex to handle in the context of introductory courses on programming. This is due to the fact that they force the novice to perform true leaps of faith in accepting an intricate syntax required even to write the simplest programs, and definitely obfuscating the underlying notional machine. Just to state an example, implementing in Java a canonical "hello, world!" requires the programmer to

1. define a class,
2. add to the latter a public, static, void method,
3. provide a contract for this method, written in terms of an array of strings, and
4. actually write the only relevant line of code, yet invoking a static method on a class variable of a system class.

of Computer Programming", a machine-oriented language, MIX (a hypothetical, "notional", assembly language). We are not proposing that every student has to learn assembly language (even Knuth [1997, p. ix] recognizes that "*is somewhat easier to write programs in higher-level programming languages, and it is considerably easier to debug the programs*"). However, we agree with Ben-Ari's advice: "*[i]n any particular course you will be teaching a specific level of abstraction* [...] *you must explicitly present a viable model one level beneath the one you are teaching*" [Ben-Ari, 2001, p. 68].

An explicit reference to the notional machine can foster meta-cognition and, during teamwork, it can help in identifying misconceptions (see 8.2.2). But how can the notional machine be made explicit? Tracing of the computational process and visualization of the execution are effective candidate tools. They allow instructors to make as clear as possible: i) what novice programmers should expect the notional machine will do and ii) what it actually does.

8.2.2 Misconceptions

Sorva defines a **misconceptions** as "*understandings that are deficient or inadequate for many practical programming contexts*" [Sorva, 2013].

Some authors [e.g., Ben-Ari, 2001] believe that computer science has an exceptional position in constructivist's view of knowledge constructed by individuals or groups rather than a copy of an ontological reality: in fact, the computer forms an "accessible ontological reality" and programming *features many concepts that are precisely defined and implemented within technical systems [...] sometimes a novice programmer "doesn't get" a concept or "gets it wrong" in a way that is not a harmless (or desirable) alternative interpretation. Incorrect and incomplete understandings of programming concepts result in unproductive programming behavior and dysfunctional programs* [Sorva, 2013].

According to Clancy [2004] there are two macro-causes of misconceptions: *over- or under-generalizing* and *a confused computational model*. High-level languages provide an abstraction on control and data, making programming simpler and more powerful, but, by contrast, hiding details of the executor to the user, who can consequently find mysterious some constructs and behaviors.

Much literature about misconceptions in CSEd can be found: we list some of the most important causes of misconceptions, experienced especially by novices, divided into different areas, found mainly in Clancy [2004], Sirkiä [2012], Sorva [2013], and in the works they reference. For a complete review see for example Qian and Lehman [2017].

English Keywords of a language do not have the same meaning in English and programming. For example, the word *while* in English indicates a constantly active test, while the construct `while` can test the condition again only at the beginning of the next iteration. Some students believe that the loop ends at the precise moment the condition is falsified. Similarly, some of them think of the `if` construct as a test continuously active and awaiting the occurrence of a condition, others believed that the `then` branch is executed as soon as the condition becomes true.

Syntax Although one may think the syntax is one of the biggest sources of misconceptions, studies show that it is a problem only in the very early stages. In particular, some students were able to write syntactically valid programs, which, however, were not useful for solving the given problem, or were semantically incorrect.

Mathematical notation Reported by many authors, classical is the confusion that generates the assignment with the = symbol (for example, seen as an equation or as a swap of values between variables) or the increment (`a = a + 1`) thought of as an impossible equation.

Examples of over-generalization Some authors found a series of non-existent constraints (e.g., methods in different classes that must have different names, arguments that can only be numbers, "dot" operator usable just in methods) dictated by the fact that the students had not seen any counterexample for such situations.

Similarities The analogy "a variable is like a box" can foster the idea that - like a box - it can contain more elements at the same time. The analogy "programming with the computer is like conversing with it" can bring to attribute *intentionality* to the computer and therefore to think that it:

- has a hidden intelligence that understands the intentions of the programmer and helps him achieve his goal (the so-called "superbug");
- has a general vision, knowing also what will happen in lines of code that it is not currently running.

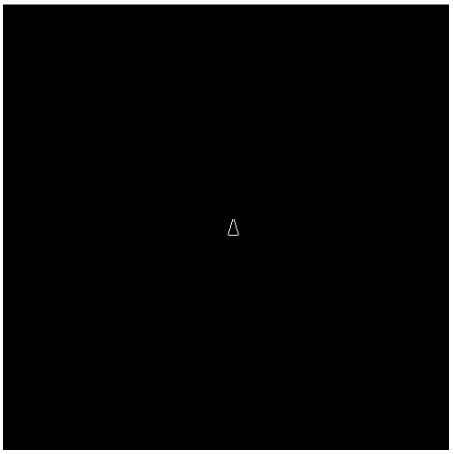
Some aspects of programming are particular carriers of misconceptions.

Sequence Many misconceptions are due to lack of understanding of the program flow: all lines active at the same time, "magic" parallelism, the unimportance of the order of instructions, difficulty in understanding the branches.

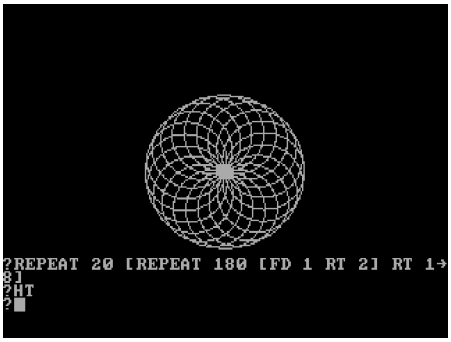
Passing parameters Students present difficulties in this area, for example by confusing the types of passing (by value, by reference ...), making mistakes with the return value or with the parameters' scope.

Input Input statements are particularly problematic. Students do not understand where the input data come from, how they are stored and made available to the program. Some of them believe that a program remembers all the values associated with a variable (its "history").

Memory allocation There are considerable difficulties in understanding the memory model of languages where allocation happens implicitly.



(a) The LOGO turtle in its home (initial) position
By Pietrodn - Own work, CC BY-SA 3.0



(b) Pattern composed of 20 circles
By Susam Pal - Own work, CC BY 4.0

Figure 8.2: LOGO programming

8.3 Past and present educational programming languages

In a constructionist view, programming languages are a key means for sharing artifacts and expressing one's theories of world. The crucial part is that artifacts can *be executed* independently from the creator: someone's (coded) mental process can become part of the experience of others, and thus criticized, improved, or adapted to a new project. As seen in Section 3.2, the origin of the notion itself of constructionism goes back to Papert's experiments with a programming environment (LOGO, see 8.3.1) designed exactly to let pupils tinker with math, geometry and physics [Papert, 1980].

In the last decades, a number of block-based visual programming tools have been introduced to help students having an easier time when first practicing programming. These tools, often based on web technologies like Adobe Flash and later JavaScript, CSS, and HTML5, as well as an increase in the availability of modern smartphones and tablets, opened up new ways for innovative coding experiences [Kahn, 2017]. In general, they focus on younger learners, support novices in their first programming steps, can be used in informal learning situations, and provide a visual/block-based programming language which allows students to recognize blocks instead of recalling syntax [Tumlin, 2017]. Many popular efforts for spreading computer science in schools, like the teaching material from Code.org rely on the use of such block based programming environments. In addition, such tools are broadly integrated in primary through secondary schools, and even at universities, in facts, they have been adopted into many computing classes all over the world [Meerbaum-Salant et al., 2013].

8.3.1 LOGO

LOGO was designed from 1967 for educational purposes by Wally Feurzeig, Seymour Papert and Cynthia Solomon [Papert, 1980]. Its syntax was heavily influenced by Lisp (at the time

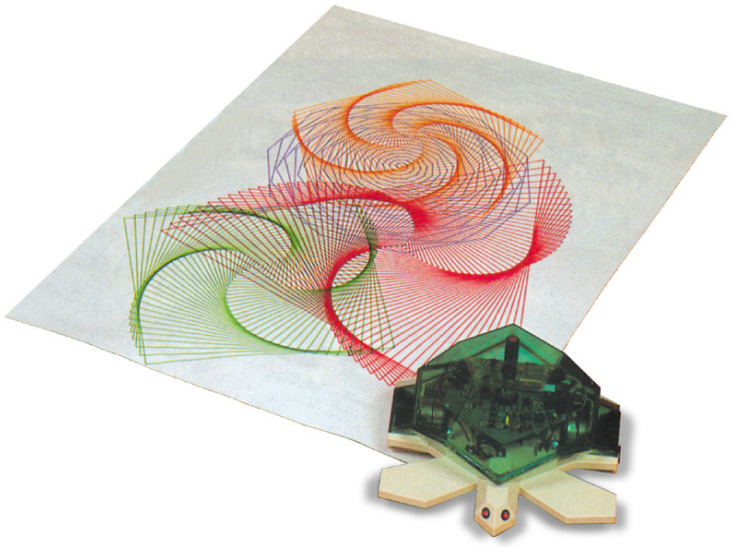


Figure 8.3: Infra red controlled Valiant Turtle Robot

the standard language for Artificial Intelligence research) and LOGO featured (among other less famous components) a graphical environment: the instructions the programmer writes are directed to a virtual "turtle" (a small isosceles triangle in which the acutest angle marks the head, see Figure 8.2a) who moves around the screen, possibly leaving a trace (see Figure 8.2b). From 1969, LOGO-controlled physical versions of the turtle were built (See Figure 8.3).

The turtle should help learners (especially the younger ones) with a sort of self-identification: its movements have a clear correspondence with our own movements in the real world, although the turtle moves in a 2D space. The patterns drawn by the turtle can be a way in which the learners build their understanding of 2D geometry, discovering in the process even deep mathematical truths as the fact that a circle (see Listing 8.1) can be approximated by a high number of straight segments [Abelson and diSessa, 1981].

Listing 8.1: A procedure to draw a circle in LOGO

```
TO CIRCLE
    REPEAT FOREVER
        [
         FORWARD 1
         RIGHT 1
        ]
```

As we will extensively discuss in Chapter 6, LOGO was originally conceived to empower learners of mathematics/geometry, not programming. Programming is *just* a means of expression, but one with a great epistemic potential. According to Papert: "*in teaching the computer how to think, children embark on an exploration about how they themselves think. The experience can be heady: Thinking about thinking turns every child into an*

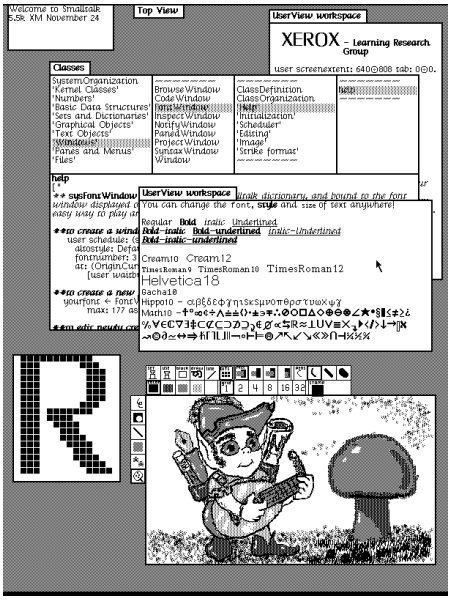


Figure 8.4: Smalltalk

epistemologist, an experience not even shared by most adults" [Papert, 1980, p. 19].

LOGO had many independent implementations and its approach is still very popular. Apart from his heir Scratch (see 8.3.6.1), many other educational languages implement this feature. Even Python[4] has a `turtle` package in its standard library.

8.3.2 Smalltalk

Smalltalk [Goldberg and Kay, 1976] also has its roots in constructionist learning. Back in the early seventies, at the Learning Research Group within the Xerox Parc Research Center, people were envisioning a world of personal computing devices which they should have intuitive user interfaces (see Figure 8.4) and an explicit "programmability". Smalltalk, with whose lineage traces clearly to LOGO and Lisp, was designed with a general audience in mind, since everyone should be comfortable with programming and computing devices should become ubiquitous in learning environments "along the lines of Montessori and Bruner" [Kay, 1993]. Thus, although clearly designed to foster a personal learning experience, Smalltalk was not directed specifically to children and it has conquered a wide professional audience.

In Smalltalk everything is an 'object' able to react to 'messages'. There follows a highly consistent object-oriented approach and code can be factored out by inheritance and dynamic binding. Smalltalk introduces also the idea that everything in the system is programmable: in fact, the tool-chain itself and the application a programmer is writing are indistinguishable and available for modifications, even at run-time. By design, such a

[4] A language not designed specifically for learning, but that is, we believe rightfully, being used worldwide as an introductory programming language.

dynamic environment encourages a trial-and-error approach. A common practice for Smalltalk developers is programming with the constant support of the debugger: instead of creating a method before its call, one sends a message that an object "does not understand", then they use the debugger to catch the exception and write the code that is needed.

A specific Smalltalk system for children was designed later as an evolution of Squeak Smalltalk: E-toys [Kay et al., 1997] provided a world of "sprites", funny characters that can be moved (concurrently) around the screen by programming them in Smalltalk. E-toys then evolved in Scratch (see 8.3.6.1), where the programming part was replaced by visual blocks.

8.3.3 Boxer

Boxer [diSessa, 1985; diSessa and Abelson, 1986] was designed by diSessa and Habelson, with the idea of creating a successor of LOGO that would stimulate the experimentation with computational literacy [diSessa, 2000; 2018]. Following the Smalltalk path on the use of graphical user interfaces, it added the idea that views everything (data and procedures) as "boxes" visible on the screen. For example, the value of a variable was visible on the screen, and the student could immediately see its state changes. The modern "variable boxes" in Scratch draw directly from Boxer, and, more generally, it paved the way for modern block languages. This is helpful also in visualizing what is going on "inside" the notional machine (see 8.2.1).

Boxer was used, for example, in teaching students important physics concepts (for example by simulating different object "falling" models diSessa [2018]).

8.3.4 BASIC, Pascal

The search for programming languages suitable for students in fields other than hard sciences and mathematics was in fact started in the sixties. In 1964 at Dartmouth College rose BASIC (Beginner's All-purpose Symbolic Instruction Code) [Kurtz, 1978]. For years BASIC has been the elective language for personal projects and even before widespread Internet connectivity, several communities shared BASIC programs in Bulletin Board Systems and magazines. Its popularity among self-taught programmers, however, was due mainly to its availability on personal and home computing devices. Moreover, the language was typically implemented using an interpreter, thus naturally fostering the trial-and-error and incremental learning styles typical of a constructionist setting. A generation grown with BASIC still thinks it is a wonderful approach to get children hooked on programming (see for example [Brin, 2016]). However, many believe BASIC is not able to foster good abstractions and fear that BASIC programmers will bring bad habits to all their future computational activities[5].

In 1970 Niklaus Wirth published Pascal [Wirth, 1993], a small, efficient ALGOL-like language intended to encourage sound programming practices using structured programming and data structuring. For about 25 years, Pascal (and its successors like TurboPascal or Modula-2) was the most popular choice for undergraduate courses and a whole generation

[5] A recent anecdote: Brian Kernighan - one of the designers of C - called a book written by a BASIC programmer "the worst C programming textbook ever written"! See https://wozniak.ca/blog/2018/06/25/Massacring-C-Pointers/index.html for the full story.

of computer scientist learned to program through its discipline of well-structured programs popularized by Wirth in his book "Algorithms + Data Structures = Programs". Only Java had a similar success in undergraduate courses. However, while Java popularity was (and is) influenced by trends in the software industry, Pascal was appealing mainly for its intrinsic discipline, which matched the academic sentiment of the time.

Structured programming had also some LOGO-like descendants, for example Karel [Pattis, 1981], in which the programmer controls a simple robot that moves in a grid of streets (left-right) and avenues (up-down). A programmer can create additional instructions by defining them in terms of the five basic instructions, and by using conditional control flow statements with environment queries.

8.3.5 Scheme, Racket

Scheme [Abelson et al., 1998] is a language originally aimed at bringing structured programming in the lands of Lisp. The language has been standardized by IEEE in 1999 and nowadays it has a wide and energetic community of users. Its importance in education, however, is chiefly related to a book, "Structure and Interpretation of Computer Programs" (SICP) [Abelson et al., 1996], which had a tremendous impact on the practice of programming education. The book derived from a semester course taught at MIT. It has the peculiarity to present programming as a way of organizing thinking and problem solving. Every detail of the Scheme (which, being a Lisp dialect, has lightweight syntax) notional machine is worked out in the book: in fact at the end, the reader should be able to understand the mechanics of a Scheme interpreter and to program one by herself (in Scheme). The book, which enjoyed widespread adoption, was originally directed to MIT undergraduates and it is certainly not suitable either for children or even adults without a scientific background: examples are often taken from college-level mathematics and physics.

A spin-off of SICP explicitly directed to learning is Racket. Born as 'PLT Scheme', one of its strength is the programming environment DrScheme [Findler et al., 2002] (now DrRacket): it supports educational scaffolding, it suggests proper documentation, and it can use different *flavours* of the language, starting from a very basic one (Beginning Student Language, it includes only notation for function definitions, function applications, and conditional expressions) to multi-paradigm dialects; this flexibility is relatively easy in a Lisp-like world, since most of the "syntax" is in fact provided by macros, that can be active or not[6]. The DrRacket approach is supported by an online book "How to design programs" (HTDP) [7] and it has been adapted to other mainstream languages, like Java [Allen et al., 2002] and Python. The availability of different languages directed to the progression of learning should help in overcoming what the DrRacket proponents identify as "the crucial problem" in the interaction between the learner and the programming environment: beginners make mistakes before they know much of the language, but development tools yet diagnose these errors as if the programmer already knew the whole notional machine. Moreover, DrRacket has a

[6] A small syntactic improvement of Racket over Lisp, very useful for beginners, is that nested parentheses can be matched easily by using different bracket families: for example, `(- {/ [* (+ 2 3) 4] 2 } 1)`

[7] http://www.htdp.org/

minimal interface aimed at not confusing novices, with just two simple interactive panes: a definitions area, and an interactions area, which allows a programmer to ask for the evaluation of expressions that may refer to the definitions. Similarly to what happens in visual languages, Racket allows for direct manipulation of sprites.

The authors of HTDP claim that "program design - but not programming - deserves the same role in a liberal-arts education as mathematics and language skills." They aim at systematically designed programs thanks to systematic thought, planning, and understanding from the very beginning, at every stage, and for every step. To this end the HTDP approach is to present "design recipes", supported by predefined scaffolding that should be iteratively refined to match the problem at hand.

The Bootstrap project[8] builds on Scheme/Racket, and has been shown to improve student performance in algebra, while engaging them in programming video games [Schanzer et al., 2018].

8.3.6 Visual Programming Languages

8.3.6.1 Scratch

EToys worlds (see 8.3.2) evolved in the currently most popular and successful visual block based programming environment: Scratch.

Scratch, launched in 2007, is a visual programming environment to create interactive media-rich projects, like video games, interactive stories, interactive art, and so on, using a set of visual blocks that represent programming instructions [Maloney et al., 2010]. Scratch was developed at MIT Media Lab by the Lifelong Kindergarten group. It is built on constructionist ideas of Papert's LOGO.

Scratch was originally written in Smalltalk, but this was hidden to most users: only a "secret" key combination can bring the Smalltalk environment alive). Version 3, released in 2019, is now build in Javascribt, and based on a fork of Google's Blockly library (see 8.3.6.2).

The Scratch site has grown to more than 47 million registered members with over 45 million Scratch projects shared programs, at October 2019[9].

Unlike traditional programming languages, which require code statements and complex syntax rules, here graphical programming blocks - that automatically snap together like Lego bricks when they make syntactical sense [Ford, 2009] - are used. In visual programming languages, a block represents a command or action. Blocks are arranged together in sequences of instructions called *scripts*. The building blocks offer the possibility, e.g., to animate different objects on the stage, thus defining their behavior. In addition to the basic control structures, there are event-triggering building blocks/conditions for event-driven programming [Fesakis and Serafeim, 2009]. Familiar concepts such as variables, variable lists, Boolean logic, user interface design, are provided as well. Thereby, these visual languages offer the same programming logic and concepts as other (text-based) programming languages. Furthermore, Scratch (like many other visual programming environments) offers the possibility to integrate graphics, animations, music, and sound to create video games, movies, and interactive stories.

[8] https://www.bootstrapworld.org
[9] https://scratch.mit.edu/statistics/

In that way, creative and artistic talents of the students are displayed in their games, stories, and applications.

The Scratch environment has some distinctive characteristics, according to its authors [Maloney et al., 2010]. Among the ones the authors highlight, some are particularly relevant in the constructionist approach:

Liveness and tinkerability The code is constantly running and can be changed on the fly, immediately seeing the runtime effects of the change; this encourages users to tinker with the code (often in a very bottom up, trial and error approach).

No error messages When you play with Lego bricks, they stack together or they don't - the same happens in Scratch; program always run: syntax errors are prevented from the block shapes and connections, and also runtime errors are avoided by doing something "reasonable" (e.g., in the case of an out-of-range value); this is particularly important not to frustrate kids and to keep them iterating and developing: "*A program that runs, even if it is not correct, feels closer to working than a program that does not run (or compile) at all*" [Maloney et al., 2010].

Other characteristics are useful to help novices avoiding misconceptions that often arise when starting to learn to program.

Execution made visible A glowing yellow border surrounds running scripts; moreover version 1.4 (and Snap!, for example) provides a "single-stepping" mode, where each block is highlighted when it is executed; this is very helpful in program reading and debugging, and helps students form a correct mental model of the notional machine underlying the program execution.

Making data concrete You can see in a variable box, automatically shown, its current value: again, this is helpful for making the underlying machine model visible.

Finally, other characteristics introduce important software engineering and development concepts.

Open source Each shared project has a "see inside" button that brings you to the project source; you can read and edit the blocks to see what happens.

Remixing If you edit someone else's project, you create a remix: you are the author, but the system automatically gives credits to the original author (at any depth, keeping track of multiple remixes in a tree) and suggests you to explicitly declare what changes you made.

One of the main critics moved by computer scientists to Scratch programs is that they do not scale well from the abstraction point of view: only since version 2 you can "make a new block" that is, a procedure with optional parameters. These blocks have no possibility to return a value (like a number or a boolean) and so can't be nested inside other blocks, forcing you to modify global variables if needed. Some on these problems are addressed e.g. in Snap! 8.3.6.2.

However Scratch (like LOGO) was not built with the intention of teaching programming concepts. Scratch main goal is, in facts, to teach digital fluency as a means of self expression rather than as a tool for future careers [Resnick et al., 2009]: often students are able to use technology as passive users, but few of them have the opportunity to be active creators through technology. Scratch belongs to MIT's vision about creative learning, and was specifically designed to help young people grow up as creative thinkers (see Section 3.3).

To help educators learn more about Scratch, computational thinking and how to teach students to be creative with programming, Harvard School of Education set up the *Creative Computing Online Workshop* [ScratchEd Team, 2013] and developed *Scratch curriculum guide* [Brennan et al., 2014]. Creative learning is explored also in the *Learning Creative Learning* course at MIT[10]. Materials of these initiatives represent the primary sources for Scratch activities described in Chapters 9 and 13.

8.3.6.2 Other Visual Languages

Snap![11] (originally BYOB, Build Your Own Blocks) is an extended reimplementation of Scratch with functions and continuations. These added capabilities make it suitable for a serious introduction to computer science for high school or college students: in fact, Snap! is used as the basis for an Advanced Placement CS course at Berkeley[12].

The Scratch approach was also ported to mainstream programming languages: in Alice [Dann et al., 2008] visual blocks are in fact Java instructions. Alice worlds are 3D: this choice makes it very attractive and appealing to pupils [Rodger et al., 2009], who can program amazing 3D animations. It also adds many complexities, since moving objects in a 3D space is not trivial.

Recently, several block-based development environments for web-browsers were published. The most popular is probably Google's Blockly[13], which allows for programming both with blocks and textual programming languages (Javascript and Python): the programmer can see the source code in different interchangeable ways.

As seen in Section 1.3.1, the no-profit organization Code.org developed teaching material made up of online interactive web tutorials, featuring famous video games and cartoons characters, highly attractive for students. The Code.org environment, *Code Studio*, is based as well on a visual programming language built with Blockly. Differently from Scratch, where a student is exposed since the beginning to the entire set of instructions and has complete freedom in the artifact to realize, in Code Studio the student is given specific tasks. The initial exercises are trivial (e.g. have a bird move straight of 2 steps) and the set of available instructions is very small (e.g. "move forward", "turn left/right") (See Fig. 8.5). Then, the difficulty degree increases slowly from one exercise to the next, and instructions and programming structures are progressively added to the set. If the student is not able to successfully complete an exercise, the system provides some standard feedback about the correctness, the number and the type of blocks used. Students are thus increasingly exposed

[10] http://learn.media.mit.edu/lcl/
[11] https://snap.berkeley.edu/
[12] https://bjc.berkeley.edu/
[13] https://developers.google.com/blockly/

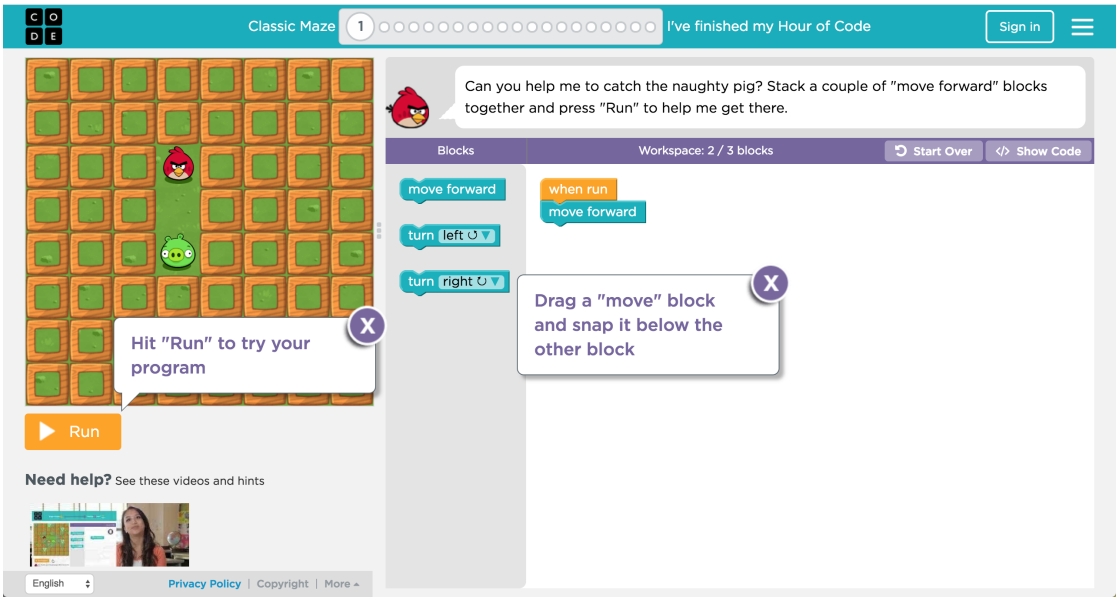


Figure 8.5: Code.org Code Studio

to the basic concepts of programming (gradually introduced while reinforcing the previous ones).

MIT has developed an environment with Blockly that can even be used to create Android applications, to be executed on mobile phones and that can take advantage of sensors and Google services like maps (App Inventor[14]). Even Apple recently proposed a block-based interface for its Swift programming language[15] and other Android-based visual programming language environments exist.

Recently, these environments evolved towards web or phone/tablet versions, in order to be available in the contexts more popular within young people (e.g., Pokect Code[16]).

All in all, visual programming languages seem to provide an easier start and a more engaging experience for learners. The ease of use, simplicity, and desirability of new visual programming environments enables young people to imagine complex goals. A study which compared three classes that used either block-based (Scratch), text-based (Java), or hybrid blocks/text (Snap!/JavaScript) programming languages showed that students generally found block-based programming to be easier than the text-based environments [Weintrop and Wilensky, 2015]. Recent studies seems to confirm that, on the long run, there is no difference in achievements and attitudes between block-based and text-based programming [Weintrop and Wilensky, 2019]. Often students only *perceive* Scratch as a mere game and, e.g., LOGO as "real" programming [Lewis, 2010; Lewis et al., 2014].

[14] http://appinventor.mit.edu
[15] https://www.apple.com/swift/playgrounds/
[16] https://www.catrobat.org/

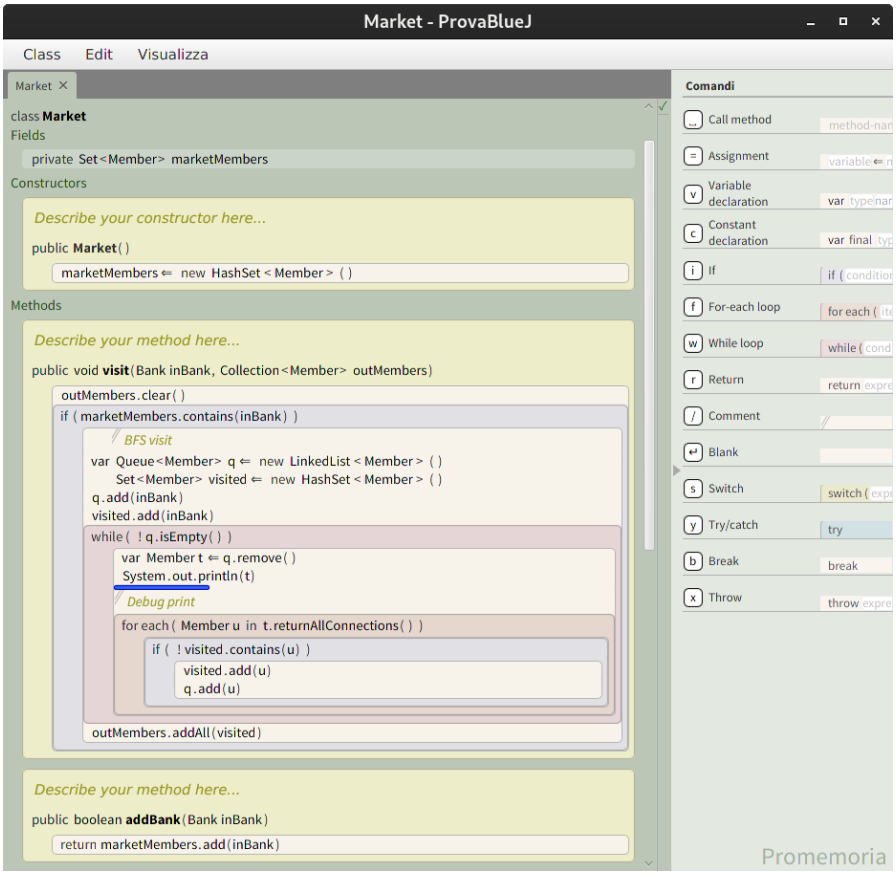


Figure 8.6: A program written with Stride frame-based environment

8.3.7 Stride

Stride [Kölling et al., 2017] is an example of a "frame-based" programming language. Frame-based editing tries to combine the advantages of text-based programming with those of the block-based programming.

Text-based systems are more readable for everyone except complete novices, have a lower "viscosity" (resistance to change), provide quicker program manipulation, more flexible navigation, especially using a keyboard.

As seen, block-based systems, among other characteristics, make many syntax errors impossible, support experimentation, make program statements more visible.

Stride is a Java-like language, integrated into the Greenfoot[17] environment and in the BlueJ[18] editor. Stride editor "*uses some graphical elements (shapes and colours) to present aspects where graphics have advantages over char- acters. Overall, however, the presentation*

[17] An environment for creating two-dimensional games and interactive simulations, using Java or Stride programming languages. https://www.greenfoot.org/

[18] A popular Java - and Stride - editor for beginners and small-scale software development. https://bluej.org/

maintains the look of a program as essentially a textual, if coloured, document" [Kölling et al., 2017], as shown in the example of Figure 8.6.

Preliminary studies reported by Kölling et al. [2017] showed that, after an initial hard learning curve, the language was appreciated by novices and experts.

We recognize that this paradigm can be useful to bring some characteristics of block-languages to a more expert audience.

In the other direction, we believe this paradigm could be a good bridge between blocks and text, and should be explored, ideally also applied to other text languages, like Python, that are considered easier than Java for novices. Moreover, one has to consider also other characteristics of environments like Scratch, that are important in a constructivist perspective, like liveness and tinkerability, the complete absence of error messages, the concretization of execution and of data, and so on (see 8.3.6.1).

8.3.8 Common features

The above short survey of programming languages for education shows they have some recurrent traits.

Personification The interpreter becomes a "persona", computation is then carried out through anthropomorphic (or, better, zoomorphic, since animals are very common) actions. This seems to contradict a famous piece of advice coming from no less than Dijkstra [1985]. Speaking of anthropomorphism in computer science, he noted that

> [t]he trouble with the metaphor is, firstly, that it invites you to identify yourself with the computational processes going on in system components and, secondly, that we see ourselves as existing in time. Consequently the use of the metaphor forces one to what we call 'operational reasoning', that is reasoning in terms of the computational processes that could take place. From a methodological point of view this is a well-identified and well-documented mistake: it induces a combinatorial explosion of the number of cases to consider and designs thus conceived are as a result full of bugs.

The *reasoning in terms of the computational processes*, however, is what is probably needed for a novice in order to familiarize with the notional machine. Some sort of operational reasoning seems also important to foster the basic intuition needed by a constructivist approach, in which concrete, public actions are key.

Visualization and tracking Computational processes that evolve in time are described by static texts: the mapping between the two is not trivial and it requires an understanding of the notional machine. Educational programming environments often try to make the mapping more explicit with some visualization of the ongoing process: the trace left by the LOGO turtle, or some other exposition of the changing state of the interpreter, for example Scratch's variable boxes.

Appeal Engagement of learners is crucial: to this end it is important to give learners powerful libraries and building blocks. It is not clear, however, how to properly balance amazing effects in order to avoid they become a major distraction: sometimes children may spend their (limited) time in changing the colors of the sprites, instead of trying to solve problems. While they surely learn about technology and creativity even in these activities, they also risk to lose their opportunity of recognizing the thrill which comes when making the computer solve computational problems for them.

8.4 (CS) Unplugged

Unplugged activities (teaching activities not using a computer / tablet / smartphone / etc.) have become popular over the years to introduce basic computer science concepts. They offer

- a constructivist environment: indeed
  - by manipulating real objects or dramatising processes, pupils can observe what happens, formulate hypotheses, validate them through experiments, i.e. develop a scientific approach to the construction of their knowledge;
  - by working in a group, pupils are encouraged to participate, share ideas, verbalize and uphold their deductions;
- inexpensive set up: they usually require very basic and inexpensive materials, so they can be easily proposed in different contexts;
- no technological hurdles: they allow students (and teachers) to have meaningful experiences related to important CS concepts (like algorithms) without having to wait until they get some technology and programming fluency.

Most famous unplugged activities are those from the CS Unplugged[19] project, started in New Zealand.

There are, however, many "unplugged" activities that aren't necessarily based on CS Unpluggd (sometimes under titles such as kinesthetic activities), with some contributed by an international community of educators, but the key elements in the approaches we are exploring here are that computers aren't required despite all of the concepts being from the computer science canon, that students are engaged in kinesthetic activities, and any equipment needed is readily available at low cost (and would often be on hand in a classroom). The term "unplugged" is sometimes used to refer to the curated activities on the open-source CS Unplugged website (csunplugged.org), but in other contexts refers to any activity relating to computer science carried out away from a computer. In this chapter we will use the full title ("CS Unplugged") to refer to the collection on the website, and "unplugged" when referring to the general concept of teaching computer science away from a computer.

The Computer Science Unplugged resources, originated in the 1990s from academics (led by Prof. Tim Bell) who had been asked to share what they did as a career with their

[19] csunplugged.org

children's peers, who at the time were around 5 or 6 years old [Bell et al., 2012]. Rather than talk *about* computer science, they chose to *do* computer science with the children, and from this point of view, CS Unplugged can be seen as an early attempt to helping students "think like computer scientists" (see Chapter 2). Ideas were taken from university courses – often advanced courses – and repackaged as physical activities where information such as graphs and binary digits were represented tangibly.

A simple example is the "parity" card trick[20], where a two-dimensional forward error correction code is introduced as a way for the presenter to somehow determine which card has been flipped over by a member of the audience. Students explore ideas for how the trick might be done, and once they discover the concept of parity, they can explore questions like whether or not two flipped cards can be identified, if it will still work with larger numbers of cards, whether a 3-dimensional version is better, and so on. Students are physically manipulating two-sided cards which (from a computer scientist's point of view) are binary digits, but for the student they need only consider their physical appearance – cards that are a different color on each side.

Another example is a game exploring routing and deadlock[21] based on passing colored objects around, with the goal of getting the correct colors to the corresponding player. The processes required to solve this quickly end up requiring backtracking and logical arguments to achieve the group's goals.

Also programming concepts can be taught with unplugged activities: in "Rescue mission" activity[22], pupils are given by the teacher a very simple language with only three commands: 1 step forward, 90 degrees left, 90 degrees right. The task is to compose a sequence of instructions to move a robot from one given cell on a grid to a given other cell. Pupils are divided into groups of three where each one has a role: either programmer, bot, or tester. The programmer must write down the instructions for the task, then pass them to the tester, who will pass them on to the bot and will observe what happens; its role is to underline what doesn't work and hand them back to the programmer, who can then find the bug and fix it.

In recent years, together with the widespread movement for CS in K-12, it is used in a variety of contexts, and with the recent adoption of elements of computer science into school curricula around the world, the Unplugged approach has often found a role in the classroom.

Tim Bell is clear in stating that CS Unplugged isn't a curriculum, and isn't intended to replace the opportunity for students to write programs on digital devices, but it is an adjunct pedagogy to enable learners to become aware of bigger ideas in computing without having the overhead of learning to program first, and also to engage in big ideas through physical movement rather than expecting all computing classes to be sitting in front of a screen. CS Unplugged is also useful for communicating succinctly to students – and more significantly, teachers and education officials – that there is a depth to computation beyond stereotypes of "coding." In a modern classroom environment, the Unplugged approach is intended to be integrated with learning to program, and this can be more effective than

[20] https://csunplugged.org/en/topics/error-detection-and-correction/unit-plan/parity-magic/

[21] https://classic.csunplugged.org/routing-and-deadlock/

[22] https://csunplugged.org/en/topics/kidbots/unit-plan/rescue-mission/

spending all of the available time on programming alone [Hermans and Aivaloglou, 2017]. When Unplugged originated, classroom computers were either too rare for students to be likely to have access to them, or the focus was on teaching students how to *use* the computer for standard productivity tasks rather than explore computational ideas with it. This situation has changed in many classrooms, and where digital devices are available, the CS Unplugged material can now be explicitly linked to programming through a "plugging it in" follow-up to the activities [Bell and Vahrenhold, 2018].

The CS Unplugged approach does not usually spell out algorithms to students, but rather, a problem is given, and students explore potential algorithms for themselves. For small instances of a problem (such as converting a number to a 4-bit binary representation, finding the shortest path in a layout with only a few vertices, or searching for an item hidden under one of a few cups) an ad-hoc or brute-force approach may find the solution easily, but as the size of the problem increases students start to encounter the need for more efficient and rigorous approaches. When asked to search for a value hidden under one of 30 cups, students often switch from a sequential search to (an approximation of) binary search, and when converting numbers to a binary representation they may discover that a greedy approach gets results, but with other challenges (such as minimal spanning trees or sorting) they may only come to appreciate that a better algorithm is needed; and for NP-complete problems, students can start to grapple the idea that no-one has (yet!) found a fast solution.

The main goal of taking a constructivist approach like this isn't that students learn particular algorithms and techniques, but that they learn that there are deep issues to be resolved in these contexts, and that they can feel empowered when they discover concepts for themselves, which can break stereotypes about what the qualities of a successful computer scientist might be.

8.4.1 Applying CS Unplugged

An important feature of this style of teaching is to give minimal instructions (often just one or two sentences are sufficient to get students started), and allow students to construct the knowledge for themselves. Once they have done this, it is important to then relate what they have done to the broader context of computing, and what happens on physical devices. Two early studies discovered that without this connection "*the program* [based on CS Unplugged] *had no statistically significant impact on student attitudes toward computer science or perceived content understanding*" [Feaster et al., 2011] and that "*the students' attitudes and intentions regarding CS did not change in the desired direction*" [Taub et al., 2012]. In terms of conveying knowledge using this approach compared with more conventional approaches, Thies and Vahrenhold [2013] found that "*it is indeed possible to weave Computer Science Unplugged activities into lower secondary computer science classes without a negative effect on factual, procedural, or conceptual knowledge*", and that it could have some benefit in that "*the Computer Science Unplugged materials can prove helpful for ability grouping within a class, since, on average, more students are enabled to reach a higher operational stage*".

Gains from using an unplugged approach were reported by Hermans and Aivaloglou [2017], who combined it with teaching programming for one group, while having a second group

spend the same total amount of time learning only programming; they found that "*the group taught using CS Unplugged material showed higher self-efficacy and used a wider vocabulary of Scratch blocks*".

Looking at these different contexts, we see that CS Unplugged is best used in combination with "plugged in" work. This is not surprising, given that getting a program to work correctly is an excellent way for a student to show that they have understood the computational concepts they are working with, since the computational agent (the computer running the program) will do exactly what the program says to do. Moreover, this will give students the opportunity to experience in a tangible (in some sense) environment the effects of their instructions, with immediate and unexceptionable feedback (rather than delayed feedback from another person, typically the teacher). Based on this, the CS Unplugged website now offers a range of "Plugging it in" exercises to provide follow-up activities that allow students to link their unplugged learning with computation on a digital device.

An unplugged approach seems to have promise for helping student learning if used effectively, but another important value of it is for teachers. Teachers need to be confident in a topic so that they can build student confidence, and given that the new computing curricula appearing around the world are often taught by people new to the subject, ways to build teacher confidence will be important [Gutiérrez and Sanders, 2009]. Often non specialist teachers can intimidated by new terminology, and looking up definitions of such terms often results in a description that is meaningless to the layperson, whereas the CS Unplugged material gives an opportunity to engage with the concept, and then learn what its name is.

CS Unplugged has been used in a variety of teacher professional development (PD) initiatives, and the research available on this is reporting positive outcomes. For example, Curzon et al. [2014] report on teacher professional development that had a substantial "unplugged" component, and noted that it was "*inspiring, confidence building and gave* [the teachers] *a greater understanding of the concepts involved*". An important feature of the constructivist approach of Unplugged activities is that they allow very quick wins, where teachers (and students) can understand a new concept (such as binary numbers) very quickly, in the context of a hands-on first-person experience, without the overhead of having to learn to program first. Smith et al. [2015] reported that teachers who were training other teachers (through the UK Master teachers system) commonly included CS Unplugged when providing professional development for colleagues, and both Morreale and Joiner [2011] and Sentance and Csizmadia [2017] found that after attending their workshop, CS Unplugged was widely adopted by teachers.

8.4.2 CS Unplugged, Constructivism and Constructionism

In K-12 Computer Science education, constructivism (and especially constructionism) might normally be associated with computer programming. Other areas such as algorithm analysis, computability, formal languages, graphics and AI could be seen as theoretical knowledge that can be acquired as needed, potentially at a later stage. However, the CS Unplugged approach inverts a traditional "programming first" view by throwing students directly into advanced concepts in topics like graph theory, error correcting codes and computational complexity. Instead of taking a theoretical approach, students are usually given a kinesthetic experience

in which they explore the issues in a way that is age-appropriate, and can engage with the ideas using a constructivist pedagogy. The purpose isn't primarily for students to acquire theoretical knowledge, but to appreciate the richness of the subject, and to give a meaningful experience those students who may not find programming to be engaging, but are interested in exploring some of the deeper issues that come up when they have access to computation through programming.

Although the unplugged approach clearly differs from Papert's constructionism because the latter recognizes programming as having a leading role as a meta-tool for constructing knowledge (See 3.2), we can point out some relationships. First of all, unplugged activities are concrete rather than formal, and aim to teach complex CS ideas to children, ideas that are usually postponed until they become adult/formal/abstract thinkers. CS concepts are not simplified, but instead made accessible with practical experiences. Second, unplugged activities generally have children using their bodies or the physical manipulation of objects to perform them.

Similarly, Papert aimed to teach deep mathematical ideas long before children had the abstraction competence to grasp them formally: "*My conjecture is that much of what we now see as too 'formal' or 'too mathematical' will be learned just as easily when children grow up in the computer-rich world of the very near future*" [Papert, 1980, p. 7]. Moreover, he designed LOGO to be "body syntonic": children could use their body to impersonate the Turtle drawing on the screen: "*working with the Turtle mobilizes the child's expertise and pleasure in motion. It draws on the child's well-established knowledge of 'body-geometry' as a starting point for the development of bridges into formal geometry*" [Papert, 1980, p. 58].

Thus, unplugged activities are a play space in which ideas from computer science can be explored, and the direction students take can be unpredictable. A constructivist approach is strongly advocated, as the primary goal is not for students to learn the concepts, but for them to discover that there are concepts that they may find interesting, and are worthy of study. Nevertheless, as computer science (and computational thinking) have started to enter school curricula, teachers have looked for ways to engage their students with specific ideas. The original activities were presented by CS researchers who were used to asking questions and exploring problems that may not have solutions, but in a classroom situation, teachers may not be experts, and aren't necessarily in a position to recognize the value of a direction a student might be taking an idea in. For this reason, the current version of CS Unplugged gives considerable guidance on scaffolding [Wood et al., 1976] the students' exploration of the ideas, and provides questions for the teacher to ask, which can create a kind of Socratic method that enables students co-construct the meaning by following fruitful paths in their exploration [Wells, 1999].

The degree of freedom left to students for exploration, and the role assumed by the teacher in guiding the activity (not as an expert delivering knowledge, but as a facilitator helping students experiment with ideas and constructing their own knowledge) plays, of course, a central role in the constructivist application of Unplugged material. This approach embraces the view that constructivism is a blend of more structured guidance and exploration, so it is not minimally guided, but *optimally* guided (See 3.1.3.1), since the teacher has a path in mind, but nevertheless, the student is constructing the knowledge for themselves, and

not having the ideas given to them directly.

8.4.3 Unplugged and transfer of CT skills

Weigend [2019] poses the question of whether or not students genuinely see the connection between the activities and computational thinking skills. It is a good point, because CS Unplugged activities alone are not guaranteed to transfer computer science concepts and attitudes.

We also believe that without explicit guidance from the teacher, it is too much to expect students to see the big picture. For example, exploring binary search can appear to focus on learning one particular algorithm and how to analyze it; however, the purpose is rather to sensitize students to the idea that the algorithm behind a program can have a significant effect on its *scalability*. If a company grows or starts processing more data, the resources required to do the computing required needs to grow only moderately as the size of the business grows. If their software innovation includes a key algorithm that is $O(n^2)$ in the number of customers, then doubling the income base will quadruple the running cost, and the process will fail due to its success in attracting interest. Issues like scalability are important, but to engage with those issues, students need to investigate particular algorithms (perhaps for searching and sorting), and in the classroom there is a risk that the learning could appear to be an exercise in memorizing a catalog of many algorithms to do the same task.

The risk is not only that students might lose track of the big picture, but teachers might also become so focused on the details of a curriculum or helping students achieve well in assessment that they put to one side thinking about why the components are there. Therefore, it is important to bear in mind the connection of computational thinking to computer science (see Chapter 6). Computer science is an independent scientific discipline with its own fundamental concepts and ideas [Bell et al., 2018] and so concepts like abstraction, decomposition, logical thinking, and so on, can and should be encountered in the context of the discipline.

Furthermore, it is strongly debated whether or not it is possible to teach general "higher order thinking skills" (see Section 4.2). We therefore agree that it is not sufficient to practice an idea (e.g., an algorithm or the idea of binary representation) in an unplugged activity to deeply learn and understand it. Similarly, in computer science education, successful transfer from solving programming problems to domain-general problem-solving skills is achieved only when activities are specifically designed to do so (see Section 4.4).

It is important to not skip the reflection phase, where students connect the activities with the computer science concepts – the recently updated website guides teachers considerably on this, and also explicitly explains the connections with computational thinking. Moreover, the new version offers "plugging it in activities" where students can use the context of programming to practice the concepts learned.

8.5 Learning to Program in Teams

Social Constructivist approaches (see 3.1.3) often emphasize the importance of social context in which the learning happens. Working in developers teams requires new skills, especially because software products (even the ones in the reach of novices) are often tangled with many dependencies and division of labour is hard: it inevitably requires appropriate communication and coordination. Therefore, it is important that novice programmers learn to program in an "organized" way, thus discovering that as a group they are able to solve more challenging and open-ended problems, maybe with interdisciplinary contributions.

To this end, agile methodologies fit well with constructivist pedagogies involving learning in teams, and they are increasingly exploited in educational settings (see for example Kastl et al. [2016] and Missiroli et al. [2016]):

- agile teams are typically small groups of 4–8 co-workers;
- agile values [Beck et al., 2001] (individuals and interactions over processes and tools; customer collaboration over contract negotiation; responding to change over following a plan; working software over comprehensive documentation) relate well with constructivist philosophies;
- agile teams are self-organizing, and emphasize the need for reflecting regularly on how to become more effective, and tune and adjust their behavior accordingly;
- typical agile techniques like pair programming, test driven development, iterative software development, continuous integration are very attractive for a learning context.

8.5.1 Iterative Software Development

For program development cycles, concepts of agile and iterative software development can be used to leverage this process and to see first results very quickly [DeMarco-Brown, 2013; Davies and Sedley, 2009].

Details about these methodologies are out of the scope of this thesis. However, we present a simplified life cycle of the process of game design in reference to agile methods, to show how a team works in iterations to deliver or release software.

The first step, *Research*, includes the development of the core idea by producing a simple game concept or a storyboard. In this phase the story, title, genre, and theme of the game should be selected, as well as a rough concept about the structure and gameplay.

In the second step, during the *Design*, the artwork, game content and other elements (characters, assets, avatars, etc.), and the whole gaming word is produced.

The *Development* phase, includes all of the actual programming, followed by *Testing* the code and the software (playtesting). Several iterations between testing and bug fixing are possible.

As a final step, the *Release* phase is planned. This could include several beta releases or a final release for end users. The agile model required to get started with the project works to bring customer satisfaction by rapid, continuous delivery of useful software.

> "In an iterative methodology, a rough version of the game is rapidly prototyped as early in the design process as possible. This prototype has none of the aesthetic trappings of the final game, but begins to define its fundamental rules and core mechanics." [Salen and Zimmerman, 2003, p. 11]

To phrase it differently, before focusing on every detail of the project, focus on the smallest step the games needs for playing it, e.g., limited interface control but basic game functionality.

Collaborative work and the connection with a real world problem makes their learning valuable [Sánchez and Olivares, 2011]. Project work is always student-centered and task oriented. The final artefacts of this project work can be shared with a community, thus fostering ownership. This collaboration is needed for developing the game idea, communicating, sharing, and managing assets and codes, playtesting and documentation.

Teachers have to consider how to support collaboration and communication during the whole game production process [Ferreira et al., 2008]. They must therefore stick to certain design patterns and iterative cycles (e.g., agile) and explain game elements and rules. The project in general should foster the teamwork in producing game assets and software. Teachers have to take into account the different preferences of students, e.g., if they feel more confident in the role of developers or artists, or that students do not shy away from switching roles. Finally, the teacher has to ensure that the ideas for the project are simple and clear, as well as reduce the size and complexity of the game projects.

8.6 Conclusions

Programming is, in some sense, intrinsically constructionist, as it always involves the production of an artifact that can be shown and shared. Of course, this does not mean that programming automatically leads to constructivist/constructionist pedagogies: in facts, we see very different approaches, from open project-based learning to more traditional education through lectures and closed exercises.

Specific languages and environments play an important role too: for example, visual programming languages make it easier (by removing the request to face unnatural textual syntactic rules) to realize small but meaningful projects, keeping students motivated, and support a constructionist approach where students are encouraged to develop and share their projects - video games, animated stories, or simulations of simple real-world phenomena. Constructionist ideas are also floating around mainstream programming practice and they are even codified in some software engineering approaches: agile methods like eXtreme Programming [Beck and Andres, 2004], for example, suggest several techniques that can be easily connected to the constructionist word of advice about discussing, sharing, and productively collaborating to build knowledge successfully together [Resnick, 1996]; moreover the incremental and iterative process of creative thinking and learning [Resnick, 2007] (see 3.3) fits well with the agile preference to "responding to change over following a plan" [Beck et al., 2001].

CS Unplugged activities can provide scaffolding to support a constructivist approach to

introducing computer science without computers, helping students construct their own initial knowledge about key ideas behind deep computational thinking concepts through kinesthetic experiences.

Nevertheless, to be effective, unplugged approaches should be used thoughtfully. The constructivist character of the activities needs to be maintained, and we know that unplugged activities are effective when used in a context where they will be ultimately linked to implementation on a digital device, either through programming, or by helping students to see where these ideas impinge on their daily life, and so explicitly fostering transfer that is otherwise improbable to happen. By using CS Unplugged early to introduce concepts, both students and teachers new to the subject can have early success without the overhead of becoming proficient enough at programming to engage properly with ideas that can have an impact in our digital world. This then provides a useful platform to motivate learning the skill of programming, but also a way to connect computer science with other subjects.

Chapter 9

Unplugged and Plugged Materials for Primary School[1]

This chapter describes some of the teaching experiences carried out by professors and researchers (including the author of this thesis) of the University of Bologna in the context of CT dissemination projects in Bologna's primary schools.

The approach characterizing the whole project is the use, either sequential or interspersed, of both "unplugged" and "plugged" approaches. The former - carried out without the use of a computer, but with materials and objects of everyday use and inspired - uses CS Unplugged activities (see section 8.4) and activities designed by us. The latter is based on Scratch visual programming language and is based, in most cases, on the *creative learning* model (see 3.3).

With respect to the Proposal for a national Informatics curriculum in the Italian school (see Chapter 7 and Appendix B), the activities cover many areas: from those more related with CS core concepts like algorithms and programming, to those more relating to computers and society, like digital awareness and digital creativity.

In this chapter, we report on two groups of activities (one "unplugged" and one "plugged") organized in sequences of lessons. In these specific activities, the author of this document had a leading role in design and implementation.

9.1 Principles

We decided to propose practical activities, in which the students are active and participate hands-on: the activities are therefore inspired, where possible, by discovery learning, and by constructivist and constructionist approaches (see sections 3.1.3 and 3.2). These activities, however, are intended to provide optimal guidance (see 3.1.3.1): children are placed in environments designed to allow them to explore and build their own knowledge, but scaffolded towards the learning objectives that we will detail.

The common thread is the dialectic between "unplugged" activities, that are carried out without the use of the computer, both "plugged," that are carried out through suitable

[1] This chapter is based on Lodi, Davoli, Montanari, and Martini [2020].

programming environments.

As described in Section 8.4.1, recent research shown that unplugged activities are especially significant when the concepts learned with them are then reused and "concretized" through the use of programming tools on a computer. Positive effects on language usage and self-efficacy have also been found.

9.2 Activities

9.2.1 Domo: a "Computational" Butler

We propose a series of unplugged activities related to algorithms and programming. The path, designed by Davoli, Lodi, and Montanari [2017a], can be freely consulted (in Italian) and modified.

It consists of a series of practical lessons aiming to introduce the basic concepts of structured programming and some simple algorithms, using unplugged activities. The course is designed for a primary class (Grade 3) that has not necessary undertaken any previous CT class, but can be adapted to each grade of primary school, as long as children have good reading skills. We tested these materials in a 3rd grade class in Bologna, during the school year 2016/2017.

The course is divided into five lessons of approximatively three hours each. The activities are designed so that the children are divided into groups, and each group is supported by a facilitator trained on the topics of the course.

Each lesson is structured with [Davoli et al., 2017a;b]:

- a brief introductory story, which places the characters in a specific problematic situation to be solved; the text can be read or narrated to the students or printed and provided to them;
- materials to be printed and cut out and provided to the students;
- a teacher guide.

Stories common theme is character who must command, through appropriately coded instructions, a robot butler ("Domo", whose name comes from the Italian for butler, *maggiordomo*) to move around on a grid and perform specific actions (e.g., picking up objects and taking them to specific cells, escaping from a maze, and so on). The grid can be printed on a sheet, and Domo represented with a pawn, or you can make the grid on the floor of the class and make one or more students play Domo. This approach is shared by other educational proposals, both unplugged and plugged (like CodyRoby[2], BeeBot[3], Code.org's Maze[4], LightBot[5], and others). The activities described here have similarities with them, but they also have some peculiarities.

[2] http://codeweek.it/cody-roby/
[3] https://www.terrapinlogo.com/products/robots/bee-bot/beebot.html
[4] https://studio.code.org/hoc/1
[5] https://lightbot.com/

- During the first lesson, students have to discover by themselves the instructions understood by the robot, and codify them with a series of symbols of their choice. This helps them discover that computers understand a finite set of instructions (e.g., the robot will understand "go ahead of one cell" but not "go ahead"), and also that the ways in which we represent these instructions are only a convention between programmer and computer, as long as there is no ambiguity about their semantics.

- To instruct Domo, the character uses cards that "physically simulate" the concept of control structures. A sequence is indicated simply by stacking the cards, while selection and iteration, which may include one or more instructions in their "body" are made with "pockets" on the structure cards in which you can insert other cards; this introduces an important physical metaphor that will then help children in structured programming.

- As one progress through the lessons, students are asked to write programs that solve not only for the specific grid situation, but are a general solution to a class of similar problems (e.g., getting out of any maze, with certain constraints); children will be encouraged to try their own solutions (e.g., on different routes of other groups). Moreover, an idea of computational complexity, in terms of the number of steps (number of cards), is given, for example, by limiting the number of cards (steps) Domo can handle.

These activities are totally hands-on: at the beginning, theoretical concepts underlying them are not explained, but students are left free to experiment. The facilitator who guides each group has the task of encouraging students to formulate hypotheses and experiment with them (just as one does when programming and "debugging").

In the first phase, the facilitator will impersonate Domo and execute, with the utmost precision and without performing *human* interpretations, the instructions that the children provide to the robot. In a second phase, he will be able to involve the children of the group to simulate the execution of the program (e.g., one will move the Domo piece - or will move on the chessboard on the floor; another will hold the cards and will read the instructions one at a time; in the case of "blocks" of nested cards within a selection or iteration, it will keep control of the condition of the iteration or the selection, and pass the cards to the partner to check the execution of that block of cards, and so on).

The facilitator will also have to manage the internal dynamics of the groups: for example, to prevent dominant personalities from monopolizing the activity, making sure that all students participate in the discussion and understand the choices made, even in the presence of particularly brilliant students.

At the end of the lesson, when the students have had practical experience of the concepts introduced, it is possible to discuss together the difficulties, things learned autonomously, and to formalize and consolidate knowledge.

The activities are highly customizable in the setting (and the teachers are invited to adapt them for the teaching of other disciplines), in the duration and in the difficulty levels.

The path is useful for reaching goals and objectives of the Proposal of national indications (see Chapter 7 and Appendix B as a reference for learning objective codes reported here in brackets) for the areas of algorithms and programming for primary school, including:

- understanding that an algorithm describes a procedure that can be automated in a precise and unambiguous manner (T-P-1);
- reading, mentally executing and debugging simple programs (T-P-2, T-P-4, O-P3-P-1, O-P5-P-1)
- breaking down a problem into smaller parts (O-P3-A-2)
- writing programs (TP-3) using sequence (O-P3-P-2), one (O-P3-P-4) or two-way (O-P5-P-5) selection, iteration for a fixed number of times (O-P3-P-3) or based on the truth of a condition (O-P5-P-2)
- understanding the concept of "block of instructions" as a single element subject to repetition or selection (O-P5-P-3)
- representation of information and data (the robot's instructions, in this case) through an arbitrary convention chosen by the students (T-P-6, O-P3-D-2)

9.2.2 Activity with Scratch

This sequence of lessons was carried out in a primary class (4th grade), that followed "Domo" lessons the previous year.

The course is partially based on constructionist and "creative computing" activities (see 8.3.6.1), designed in the context of creative learning (see 3.3), and partially contains more structured and guided activities, still following a constructivist approach (see 3.1.3).

The primary tool for this approach is the Scratch programming language (see 8.3.6.1).

The course has been realized in three lessons of three hours each, but it is highly customizable through materials available online.

Students must have computers or tablets able to run Scratch. They can work alone if the school is equipped with a computer for each student, or in pairs.

The workshop is held by a conductor, who introduces the concepts and challenges, with the help of one or two other facilitators who - together with the conductor - walk around and support students in the realization of projects.

The conductor creates a "Teacher account"[6] on Scratch and then creates a class, assigning the students a username and a password that would not allow their identification (since even the student accounts on Scratch are public - even though only for the limited time in which one decides to keep "open" the class).

Lesson topics are detailed below.

Lesson 1

- Scratch surprise [Brennan et al., 2014]: after a concise (2-3 minutes) introduction to the Scratch editor interface, students are given time to explore the potential of the tool, and are asked to try making the cat do something amazing.

[6] https://scratch.mit.edu/educators#teacher-accounts

- Login into the virtual class.
- Tutorial: the teacher guides the students in building a project step by step[7], progressively introducing features while discussing with students, and letting them discover some part of the solution. The game mechanics (and therefore the code) will be substantially the same for everyone, but students are strongly encouraged to customize the setting and to add features once the tutorial is finished. The tutorial contains most of the basic elements of structured programming (sequence, selection, iteration, variables, and even events). To facilitate the transfer, during the lesson, we explicitly recalled the actions that Domo could perform, asking students to recognize how they match with Scratch blocks. It is also important to show students other examples of projects that can be realized with Scratch[8] (e.g., interactive stories, art, simulations) to prevent them from associating the tool exclusively with video games.
- Saving and sharing: it is essential to explain to students that they are encouraged to share their projects, even if incomplete, to receive feedback from other users. They are also invited to write a description and the instructions for the game, and to give credits (e.g., ideas, collaboration with partner, media taken from the web) in the appropriate areas on the sharing page. This point is particularly important for introducing students to the idea of "open source code", at the basis of free software: in fact all projects shared on Scratch can be "seen inside" to study how they were created, and "remixed", i.e., you can create a new project that makes use of the code of others (automatically giving credits).
- Insertion of projects in a "class gallery," which helps everyone to identify and explore the projects of their classmates.

Lesson 2

During the first lesson, many students, encouraged by Scratch's natural predisposition to be freely explored, expressed their willingness to create their own free projects. Therefore, we dedicated the second lesson to the creation of personal projects. Students were given more than two hours to build a personal project. In order to encourage students to have a "presentable" project at the end of time, motivate them, and allow them to receive feedback from their classmates, students were told that everyone would show their work to the entire class at the end of the lesson.

In such open situations, some students will already have ideas to do their work, while others will be displaced by such freedom. This is why the students were suggested to look for help on Scratch and to explore other projects. To facilitate "stuck" students, we provided so-called "Scratch cards," which give ideas or help implement a specific feature (e.g., make a character change color). Those we used are particularly lighweight[9], while others guide step

[7] Inspired by Carmelo Presicce's "Under the sea": https://scratch.mit.edu/projects/14759947/
[8] https://scratch.mit.edu/explore/projects/all
[9] https://goo.gl/ffGX1J

by step in the realization of an initial project[10].

The role of the facilitators was to encourage, to help solve specific problems (though trying not to direct students towards a specific solution), but also to help students design their own artifact so that the result was creative but also feasible within the time constraints.

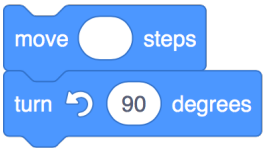
During the presentation, it was essential to stimulate reflection, asking questions about the difficulties encountered, about the things that one had learned, about what students wanted to add, and stimulating other children to make comments or criticisms.

Lesson 3

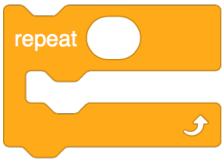
In the third lesson, we used Scratch to teach some geometric concepts. In particular, the final objective was to create a general program which, given any $n > 2$, draws the regular polygon with n sides. Students should first experiment with the "pen" function of Scratch, which - in analogy with the LOGO turtle (see 8.3.1), allows a sprite to "leave a trace" when moving, i.e., drawing.

First of all, students are asked to draw a square, possibly helping them by physically mimicking a person who walks a certain number of steps, then turns 90 degrees and so on 4 times.

If the students have repeated four times the pair of instructions



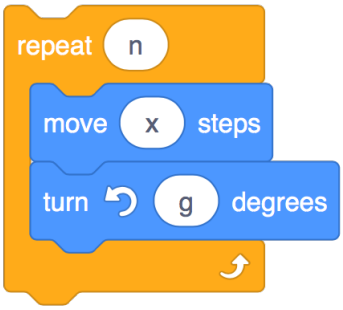
it is possible to point out the possibility to use a repeat block



for example by asking them what they would have done if they had been asked to draw a polygon with 20 sides, and making them reflect on the fact that by entering the movement instruction only once, they could change the number of steps (and therefore the length of the side) by changing this value only once.

At this point, the students are asked to draw a triangle. The typical mistake is to turn the sprite 60 degrees (internal angle) and not 120 degrees (external angle). Students can be helped once again with human dramatization or by drawing on the blackboard. At this point, students are asked to continue with the other regular polygons (pentagon, hexagon, etc.), and they should note that the program structure is always the same:

[10] https://scratch.mit.edu/ideas



where (n) is the number of the sides of the polygon, (x) is the length of a side, and (g) the exterior angle. We can point out to students that that (g) decreases with increasing sides (Table 9.1).

Table 9.1: Relation between sides and turning angle

sides (n)	**degrees (g)** (exterior angle)
3	120
4	90
5	72
6	60
...	...

The goal is to make students realize that a regular polygon will have all the external angles equal and that their sum must necessarily be 360 degrees, in order for the polygon to "close" perfectly.

In this way, we are trying to teach a rule constructively: "The sum of exterior angles in a polygon is always equal to 360 degrees.", making the students experience the consequences of this rule before it is stated.

Digital Awareness

During the lessons, the students were invited to visit the Scratch site at home, as well. Scratch is a safe community for children because it is extremely moderated by the Scratch Team. Furthermore, with a teacher account, the teacher can check and delete inappropriate content or comments. The projects are, however, public and can be commented on by the whole community (but there is no way to exchange private messages). Pupils were therefore asked not to give personal information and to report anything suspicious to the instructors. Some basic social behavior rules have also been taught (e.g., avoiding offensive comments). Critical was the reading and discussion of the "Scratch community guidelines"[11], rules of conduct valid well beyond the platform itself. These aspects were also discussed with parents who,

[11] https://scratch.mit.edu/community_guidelines

initially fearful, then understood the importance of introducing - through a highly controlled and educational community - their children to the conscious use of social media, on which they will, soon or later, find themselves.

This activity covers many points of our curriculum proposal (see Chapter 7 and Appendix B). The most important goals that these activities aim to achieve are obviously linked to the expression of algorithms through programs in a certain language (T-P-2) and in general to the ability to write of simple programs (T-P-3). Some activities are focused on stimulating the free creative expression of children, and therefore tend to personalize learning: different students may have learned, at different levels, a vast range of different concepts. These activities can be followed by more structured ones, to make the whole class achieve certain specific objectives in the areas of algorithms and programming. The strength of these activities is, in fact, to let children experience the possibility of using computer applications as a tool for personal expression (T-P-11) and, specifically, to create and express themselves through the creation of programs (stories, games, ...) (O-P5-R-2). The discussion and attention to the rules of conduct in the Scratch community also help achieve goals and objectives in the context of conscious use of new technologies, including:

- rules for the safe and responsible use of technologies (T-P-9),
- recognition of information privacy and confidentiality issues (O-P3-N-2),
- respect for others (O-P3-N-3),
- acceptable / unacceptable behavior in the use of information technology and of content obtained through it (O-P5-N-4),
- request for help from adults (O-P5-N-5).

9.3 Conclusion

This chapter shows examples of activities that:

- let students construct knowledge about fundamental concepts of structured programming (e.g., sequence, conditionals, loops, variables) but also complexity in terms of computational steps and generalization of algorithms through unplugged activities structured as an incremental discovery, scaffolded by the instructors;
- then propose plugged activities that follow the creative learning approach, using Scratch as the main tool, both for free creative expression and for learning other disciplines (e.g., drawing regular polygons);
- let student increase the awareness of rules and threats of a digital community by experimenting in a safe one (the Scatch community);
- are designed to achieve the competence and knowledge goals presented in our curriculum proposal (Chapter 7).

The proposed activities must be seen as a proof of concept, to inspire the building of teaching activities in this direction. Of course, the activities need scientific validation and measurement to verify that they help achieve the suggested learning objectives.

During the first implementation, we collected only anecdotal evidence, that, however, is useful as a starting point for more precise research.

In particular, during the hours dedicated to "Domo, a computational butler," we found that children were able to solve increasingly complex problems with increased mastery. The possibility of physically testing their solutions - by simulating, in turn, the execution of the program - was particularly exciting for the children. We believe the ability to find computational solutions - learned through the proposed unplugged activities - was successfully applied to the resolution of problems through Scratch. The unplugged activities allowed children to internalize some fundamental concepts for the formulation of algorithms. This allowed them to orient themselves with the Scratch environment quickly, easily identifying (also thanks to explicit teacher prompts) and using the same structures to solve the proposed challenges or to create games, stories, and animations.

The primary teacher supporting the university instructors particularly noted children's' willingness to collaborate, especially in new activities compared to the usual didactic context, a context in which they tend to work individually to pursue their own personal objectives. Initially, the topic appeared difficult for the teacher herself, but very soon, she found the absolute openness of her students to new experiences. The fact that the activity was presented as a practical, hands-on activity - whit reflection phase coming only after the practical implementation - was very much appreciated by the teacher. She also noted that, in a context so different from the usual classroom practices, some of the apparently weaker children were able to give answers and find solutions for their group, gaining great satisfaction.

Part III

Teachers' Conceptions

Chapter 10

Sentiment About Programma Il Futuro Project[1]

In this chapter, we report a 2016 research, where the teachers' sentiment after the first two years of "Programma il Futuro" (PiF) project were analyzed.

As described in Subsection 1.4.5, the project is the Italian localization of Code.org with a support website, providing guidance, teaching materials, tutorials, forums.

Almost all students found the material useful and were interested, teachers have reported. They have also declared to have experienced high satisfaction and a low level of difficulty.

A detailed analysis of quantitative and qualitative answers about the project is presented and areas for improvement are identified.

One of the most interesting observations appears to corroborate the hypothesis that an exposure to informatics since the early age is important to attract students independently from their gender.

By contrast, most of the positive outcomes envisaged by the teachers were more related to engagement and transversal competences, than on specific learning of CS concepts.

10.1 Participation Data

Since the first year (school-year 2014-15), Italian teachers have been highly reactive to the initiative, making Italy the most active non-english speaking country for what regards informatics education in school, at least in terms of participation to the CSEd week[2].At the end of school-year 2015-16 about 14,000 teachers in more than 4,000 schools had involved more than 1 million students (about one eighth of Italian students) in the activities. Project participation has tripled from the first to the second school-year. Details of participation during the first two years are described by the four charts in Figure 10.1, showing trends for schools, teachers, classes, and students. Updated data are reported in Subsection 1.4.5.

[1] This chapter is based on material published in Corradini, Lodi, and Nardelli [2017a].

[2] https://hourofcode.com

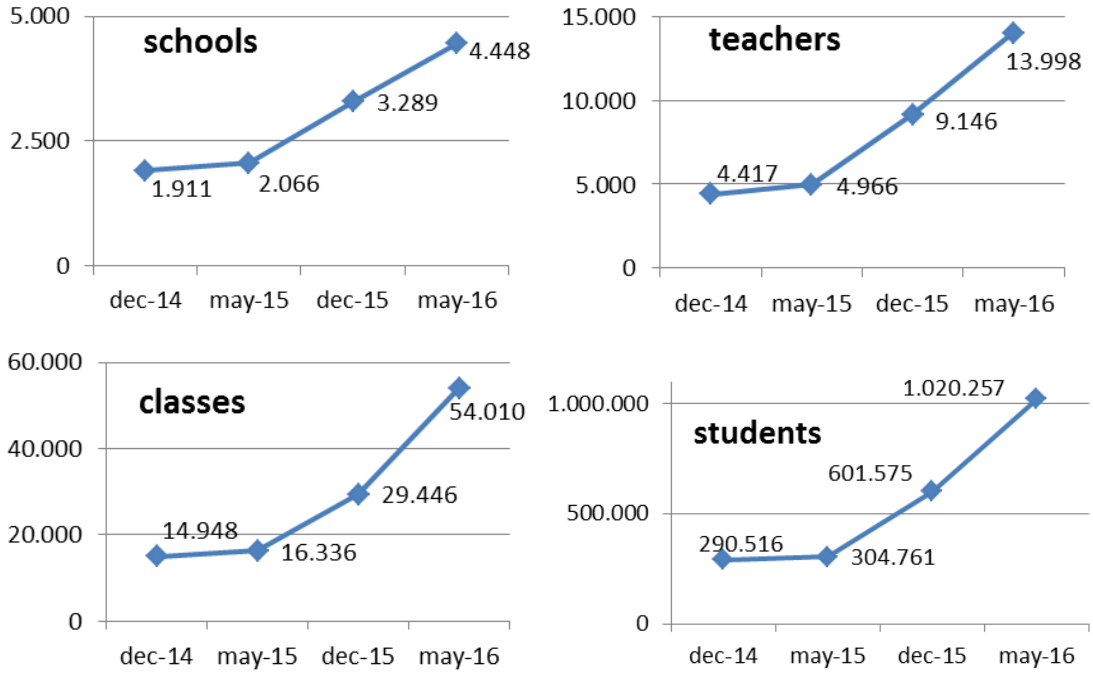


Figure 10.1: Students, teachers, classes and students participation.

10.2 Project Monitoring

10.2.1 Data Collection

The PiF project monitors progresses two times a year through a questionnaire sent to all teachers, first in December, right after the CSEd week, and then in May, a few weeks before the end of school year[3].

The questionnaire collects descriptive data about teachers and their classes and schools, quantitative data about students participation to coding activities, and qualitative feedback. A few optional questions are open and are intended to investigate both positive and negative sentiments of teachers with respect to the project.

The percentage of answers received has always been high (15% to 17% for 2014-15; 21% to 24% for 2015-16; number of recipients for each of the four questionnaires is shown in graph "teachers" in figure 10.1), providing a good confidence that values and comments received are reasonably representative of the situation of the entire population.

10.2.2 Quantitative Data Analysis

In school-year 2015-16, more than half of teachers was in primary school, and almost a third in lower secondary (see Fig. 10.2) while in the previous year there was a higher percentage of

[3] http://programmailfuturo.it/progetto/monitoraggio-del-progetto

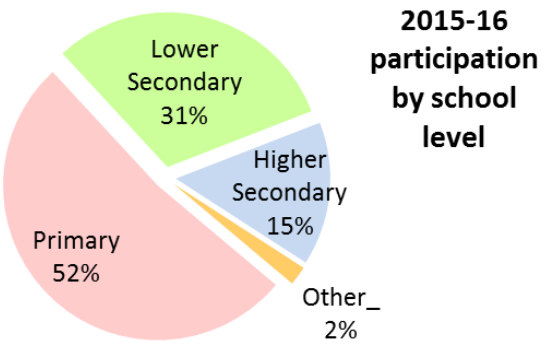


Figure 10.2: Participation by school level.

primary school teachers (56%) at the expense of lower secondary one (27%).

It is interesting to observe the different distribution of subjects taught by the teachers involved in the project according to the different level of school. Subjects have been classified in two large groups: literary and scientific/technical, while informatics has been considered on its own. Please recall (Sec. 1.4.2) that both in primary and lower secondary school generally informatics is not an independent subject. The distribution (shown in the two charts in Fig. 10.3) does not significantly change between the two school-years. It is a highly positive element the fact that also teachers of literary subjects have involved themselves in bringing computational thinking to the attention of their students.

It was asked to teachers to evaluate how useful was the activity for their students on a 4-point Likert scale: 98% of them answered "useful" or "very useful". It was also asked them to evaluate how interested were their students during activities: 98% of them answered "interested" or "very interested". These outcomes are essentially the same in the two school-years.

Teachers were asked to evaluate whether, in their classes, students were equally interested by the activities irrespective of their gender, or females/males were more interested. Results, shown in the chart in Fig. 10.4, are similar across the two school-years and exhibit an increasing polarization when students grow up.

Similarly, teachers were asked to evaluate effectiveness of students in executing activities with respect to their gender. Also in this case the results, shown in the chart in Fig. 10.5, are similar across the two school-years: the older the students are, the higher is the polarization.

We think both results, hinting that informatics acceptance has a higher independence from gender when pupils are younger, provide some support for the importance of exposing students to it at an early age.

10.2.3 Qualitative Data Analysis

We now discuss the sentiment analysis regarding teachers' answers to open questions.

Positive sentiments were explored by two open questions: "Describe the most positive

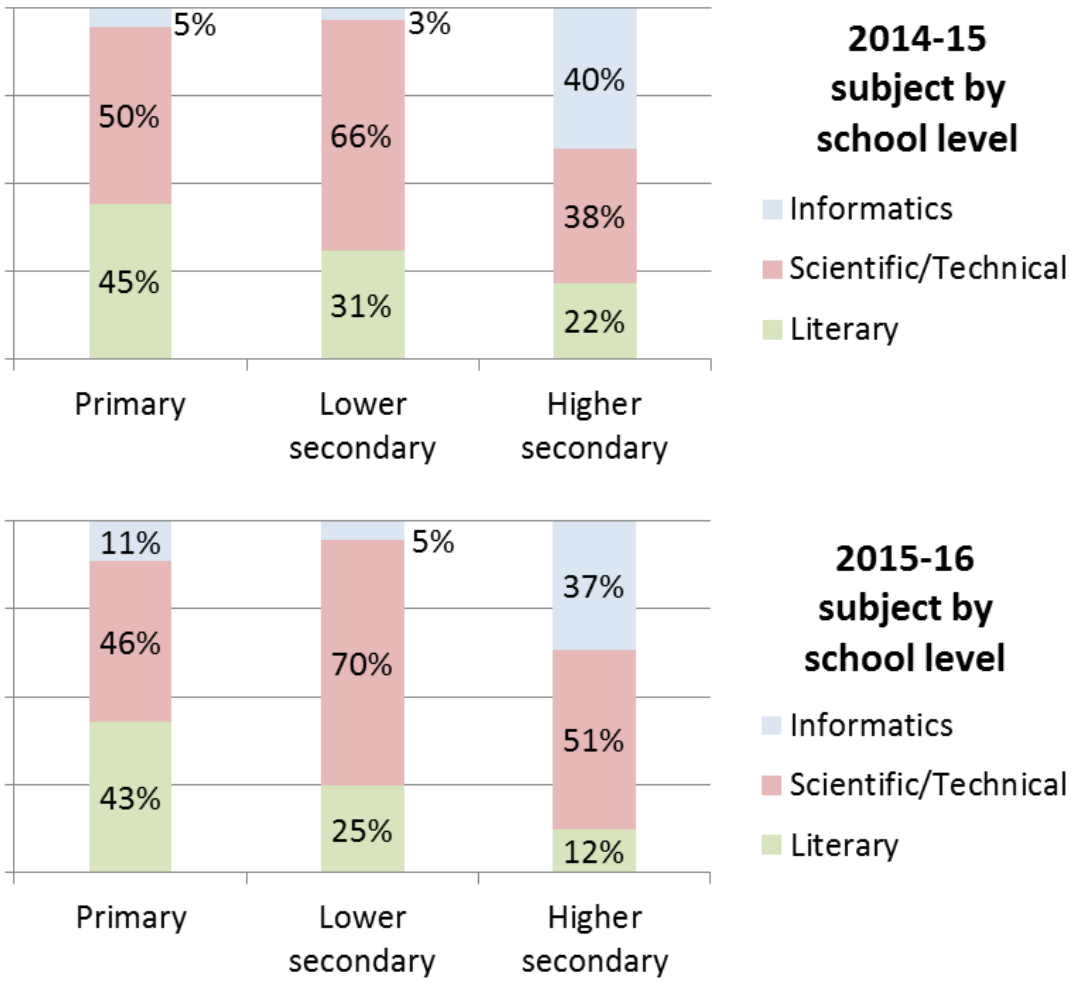


Figure 10.3: Teachers' subject distribution by school level.

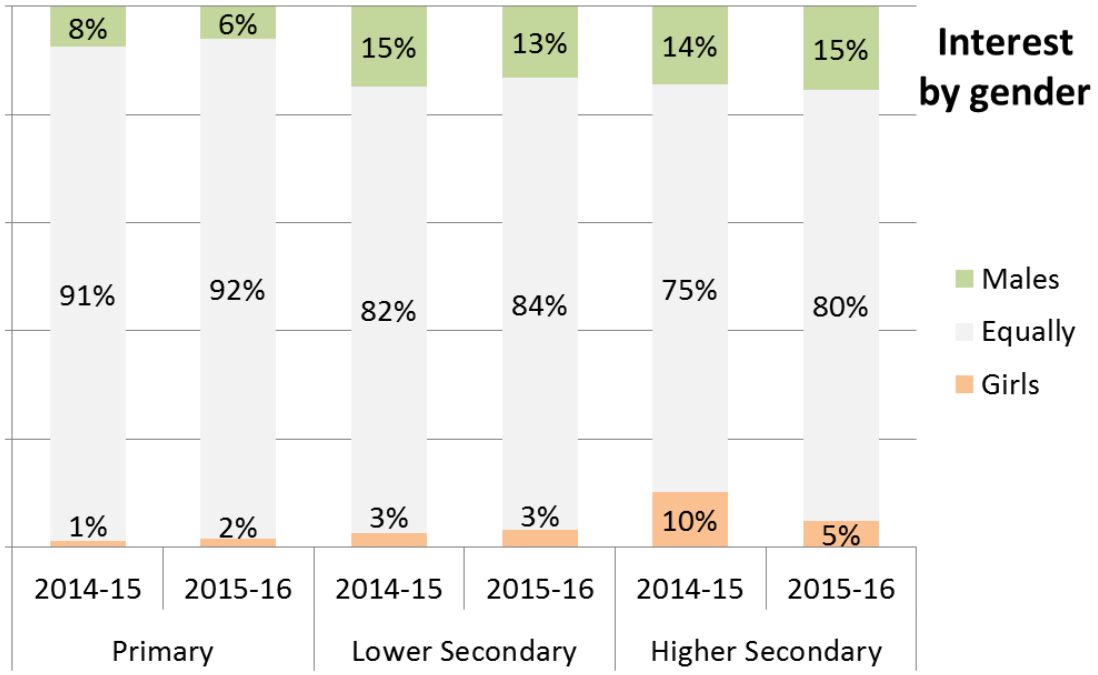


Figure 10.4: Interest by gender.

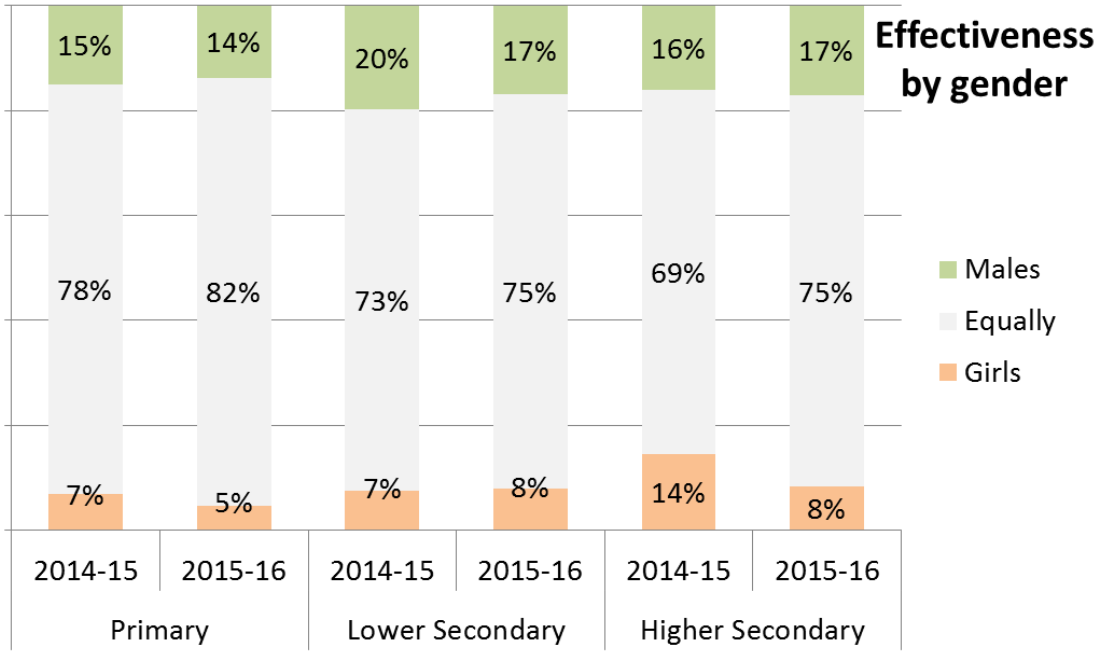


Figure 10.5: Effectiveness by gender.

Table 10.1: Cluster distribution of positive sentiment answers

	Positive factors	Reasons to suggest
Cognitive	20%	26%
Motivation	32%	26%
Methodological	20%	26%
Quality	18%	9%

factors in the project" and "Provide a reason to suggest participation to a colleague". A total of 1,342 (resp. 1,313) answers, across the two school-years, have been provided to the first (resp. second) question.

In a first phase, all answers were processed collectively by the three researchers involved (among which there is the author of this dissertation), and divided if they contained more than one concept. We thus obtained a total of 1,523 (resp. 1,551) single concept sentences. All sentences were analysed again by the researchers to identify recurring themes, that were found to be common to both groups of answers. These became the clusters used to classify sentences. Finally, each single concept sentence was manually assigned to one of the clusters. Here is the short name and description of the most relevant ones:

- *Cognitive stimulation and cognitive development* (promotion of awareness and comprehension of: computational thinking, problem solving, logical thinking, creativity, attention, planning ability, ...)
- *Motivation and participation* (motivation for learning, students interest, students and teachers involvement, cooperation between students)
- *Methodological aspects* (effective outcomes, ludic learning, innovative approach for teaching informatics, inclusive didactics)
- *Quality of instructional material* (well prepared, attractive, structured for gradual learning)

In both groups (see Table 10.1) *Motivation and participation* is the most frequent cluster, while also *Cognitive stimulation and cognitive development* and *Methodological aspects* play an important role. Cluster *Quality of instructional material* is perceived more as a "positive factor" then as a "reason to suggest", which is a viewpoint coherent with teachers' pedagogical perspective.

Other interesting answers by teachers, not included in the most relevant clusters, warrant deeper consideration and analysis in further works. For example, some teachers stressed the positive consequences in terms of attention improvement for students with concentration difficulties.

Negative sentiments were first explored by means of a follow-up open question to a yes/no question: "Have you experienced difficulties?". Furthermore, a yes/partly/no question: "Has the project matched your expectations?" is followed with an open question asking for

clarifications, in case of partly or total mismatch. A total of 334 (resp. 275) answers, across the two school-years, have been provided to the first (resp. second) question.

The lower number of answers to these questions (roughly speaking, the total number of negative remarks is about one quarter of the total number of positive ones) can be a clear indication of the general satisfaction of teachers with project activities. Indeed, 91% of teachers did not report any difficulty during school-year 2015-16 (it was 88% in 2014-15), and 84% of teachers were fully satisfied in 2015-16 (82% in 2014-15).

A first analysis of answers to the two open questions investigating negative sentiments showed that the kind of remarks were similar. It was therefore decided to merge the two sets and carry out a cluster analysis on the whole set, using the same methodological approach used for positive sentiment analysis. Multiple concept answers were simplified in 786 single concept sentences, that were manually partitioned in disjoint clusters.

We now list the most relevant topics resulting from this analysis, with a short name, a description and its ratio in the overall distribution:

- *Technical problems* (34%) (Obsolete or too few devices, absent or very slow Internet connection, ...)
- *Teacher training* (18%) (Lack of personal knowledge to solve the exercises or to prepare an adequate lesson plan with specific computational thinking learning objectives, too little time to self-train, absence of specific training courses, difficulties with English-written material, ...)
- *Organizational and logistic problems* (16%) (Mainly lack of time to teach the material during lessons due to an already crowded school schedule)

All these clusters point to infrastructural problems, that are independent from scientific issues concerning informatics education.

Other clusters, with a lower ratio, can provide useful hints for actions aiming at introducing informatics education at all school levels in Italy. In particular we found other four main topics:

- *Limitations of platform and support site* (11%) (Both technical problems or lack of features of Code.org and problems with the support site, sometimes stated as not clear or too verbose)
- *Quality/level of teaching material* (10%) (Material too easy or too difficult - and so not engaging - for the specific age level of the students)
- *Curriculum and didactics* (6%) (Teaching effects not clear or visible, difficulties in integration with standard curriculum, lack of creativity in activities - often compared to Scratch)
- *Colleagues/parents involvement and support* (5%) (Lack of support from colleagues during the activities or from parents at home)

Issues related to *Curriculum and didactics* are the most relevant ones to move from a stimulus action phase to a full operational one.

A final open optional question asked for "Observations and suggestions": most of its answers have been positive, stressing the importance of computational thinking education in Italian schools and showing willingness to continue activities, even proceeding in autonomy. The main request for improvement has been to provide the italian dubbing of videos accompanying the courses.

10.3 Conclusions and Future Perspectives

Outcomes of project monitoring show that informatics education is a highly interesting theme for both teachers and students and that in a short time span the proposed teaching material has been adopted, used, and appreciated in a significant number of classes. We therefore think that appropriateness of teaching material is a key factor to bring informatics education in schools. Hence the project is on track to meet the goal of spreading among school teachers more awareness of the principles, concepts and methods of informatics.

Concerning competence acquisition by students, we do not have a formal measure of their progresses in the project, since the Code.org material does not include a set of assessment tools. Clearly, since learning material is partitioned into very small chunks and later exercises require having learned previous concepts and skills, progressing in courses is a good proxy indication of actual competence acquisition. How to carry out in schools the measurement of the acquisition of the various informatics competences is an open problem, to be tackled by joint efforts by computer scientists and pedagogists.

Another important issue, raised by some teachers, is how to merge the "closed" teaching paths provided by Code.org material with the need of providing more "open" venues, where both teachers and students are able to give space to their creativity. This issue, and the more general one whether it is better a "puzzle based" or a "project based" approach to informatics education, is also highly debated in the research community [Resnick and Siegel, 2015].

Teachers' answers corroborate the urgent need of teacher training in CS teaching and CS basics as well, both because this is asked explicitly, and because we see a tendency to recognize the value of the project more in fostering transversal competences or domain-general higher order skills (like promotion of awareness and comprehension of problem solving, logical thinking, creativity, attention, planning ability, motivation for learning, students interest, cooperation) than in teaching CS core concepts.

During following years, the project evolved by taking into account some of these issues (for example by promoting creativity contests and professional development initiatives - see 1.4.5).

Of course, the project must be seen as a facilitation, to foster a more institutional and systemic introduction of CS in school curricula (see Chapter 7).

Finally, we found that teachers perceived that the interest in their students with respect to gender is equally distributed in primary school, but tend to increase for boys and decrease for girls from secondary school. This reinforces the idea that it is essential to introduce CS from primary school, when gender stereotypes are not yet present.

Chapter 11

Conceptions and Misconceptions About Computational Thinking and Coding[1]

As seen in Chapter 1, many advanced countries are recognizing more and more the importance of teaching computing, in some cases even as early as in primary school. "Computational thinking" is the term often used to denote the conceptual core of computer science or "the way a computer scientist thinks", as Wing put it. Such term - given also the lack of a widely accepted definition - has become a "buzzword" meaning different things to different people (see Chapter 2).

In this chapter, we report the results of a large scale (N=972) investigation on Italian primary school teachers, conducted in the context of "Programma il Futuro" project (§ 1.4.5).

First of all, we investigated the teachers' conceptions about computational thinking. Teachers have been asked to provide a definition of computational thinking and to answer three additional related closed-ended questions. The analysis shows that, while almost half of teachers (43.4%) have included in their definitions some fundamental elements of computational thinking, very few (10.8%) have been able to provide an acceptably complete definition. On a more positive note, the majority is aware that computational thinking is not characterized by the use of information technology.

Secondly, we are aware that the CT movement has popularized the use of the term "coding". While computing professionals and academics tend to use it to denote a part of or even the entire process of "programming", policy makers, media, and some instructors often use "coding" to indicate something new and distinct from "programming" in professional or scientific sense (§ 2.4). For this reason, we explored teachers' views on the terms "coding" and "programming", and how they are related to their ideas on "computational thinking". When directly asked "if coding is different from writing programs", roughly 2 out of 3 teachers answered "no". Among the teachers who answered "yes", almost 160 tried to motivate the difference: a few of them gave admissible explanations, while the others showed various

[1] This chapter is based on results published by Corradini, Lodi, and Nardelli [2017b; 2018b].

misunderstandings, which we classify and discuss. By contrast, when asked about their idea of "what coding is", only 4 out of 10 of the teachers explicitly linked coding to programming, but an additional 2 out of 10 cited an information processing agent executing instructions. The remaining part of the sample did not provide explicit or implicit links between coding and programming.

Our investigation shows that untrained teachers hold misconceptions regarding CS and its related terms. Given the general public and media attention on "coding" in schools, currently taught by existing teachers - mostly not appropriately trained, professional development actions focusing on CS scientific principles and methods are therefore a top priority for the effectiveness of CS education in schools.

In this chapter we use the term *misconception*. In general, the term indicates an incorrect view based on faulty thinking or understanding[2]. In Computer Science Education research literature, the term is often used in the specific context of learning to program, and refers to an inadequate understanding of fundamental programming concepts (see § 8.2.2; for a complete review see Sorva [2013]). In this chapter we are not referring to such difficulties, but rather to incorrect ideas about CT and "coding"; so we are using the term in its general sense (like, e.g., in Denning et al. [2017]).

11.1 Purpose of the Study

Our research has two main objectives. In the first part, we investigated the knowledge level of CT among Italian primary school teachers. More specifically, we addressed the following research questions:

RQ1 which level of understanding do they have with respect to the concept of computational thinking?

RQ2 how do they perceive the relation between technology and computational thinking?

RQ3 how much do they feel prepared to teach computational thinking?

In the second part, we investigated how Italian primary school teachers define *coding* and which relations they see between it and *programming*. More specifically we addressed the following research questions:

RQ4 how do they define coding?

RQ5 how do they perceive the relation between coding and writing programs?

11.2 Related Work

A few works investigated teachers' conceptions about CT, computing and their relation with IT.

[2]Oxford Dictionary, https://en.oxforddictionaries.com/definition/misconception

Yadav et al. [2011a;b] conducted two experiments to asses pre-service teachers' "attitudes towards and understanding of computational thinking" and how they changed after attending a CT module in a course of an Education major. Both a pre and a post questionnaire were used in these studies. The first one (N=100) did not have a control group, that was introduced in the second study (N = 294, 141 in the treatment group and 153 in the control group). In both experiments, results showed such module was effective to influence teachers' understanding of CT and to improve their positive attitudes toward CT and its integration into the classroom.

Bower and Falkner [2015] conducted a pilot survey on 44 pre-service teachers, investigating their awareness of CT, conceptions regarding the term, use of IT and pedagogical strategies for CT development, and confidence in teaching CT.

Duncan et al. [2017] report the post-lesson feedbacks from 13 primary school teachers (with no previous experience in teaching computer science) participating in an ongoing study on teaching CT in New Zealand. They report about teacher confidence, level of difficulty of the lessons, common themes emerged in the answers, and teachers' misconceptions.

We were not able to find studies specifically investigating teachers' ideas about the relationship between *coding* and *programming*.

11.3 Methods

11.3.1 Instrument

As described in Chapter 10, periodical surveys are conducted in "Programma il Futuro" project by means of on-line questionnaires collecting quantitative and qualitative data.

We investigated our research questions in this context. A questionnaire, with some additional questions relevant to the current research, was sent in December 2016, after the CS Educational Week, to all 24,939 teachers enrolled into the project. They filled it out anonymously and we received 3,593 answers up to the end of January 2017.

Teachers belong to all level of schools, from kindergarten to higher secondary schools. Some of them participated to the project for the first time in school-year 2016/2017, others for the second or third time.

In the first part of the questionnaire, we asked teachers to complete - if they wished - the sentence

Q1 "*In my view computational thinking is...*"

that is we asked them to provide their definition of CT.

We also asked teachers to choose their level of agreement, on a 4-point Likert scale, with the following statements:

Q2 *Being able to use technological devices means having developed computational thinking competences*

Q3 *Computational thinking competences can be adequately developed in primary schools without using technological devices*

We finally asked teachers to grade (4-point Likert scale)

Q4 *How much do you feel prepared to develop computational thinking in your students?*

and to indicate the

Q5 *Most important initiatives to improve your preparation*

by choosing up to 3 answers among:

- training
- availability of technology
- organizational support
- methodological guidelines
- learning objectives and teaching content

In the second part of the questionnaire, we focused on teachers' answers to the following questions (in square brackets the actual Italian wording, to stress the fact that the word *coding* is untranslated, as usual for this term in Italy).

The first one asked them to provide their definition of coding by completing:

Q6 *In your view coding is...* [Secondo te fare coding è...]

The second one asked teachers to answer:

Q7 *In your view is there any difference between coding and writing programs?* [Secondo te c'è differenza tra "fare coding" e "scrivere programmi"?]

and to those answering positively it was asked:

Q8 *If you wish, explain why* [Se vuoi, spiega perché:]

In the current study we focus only on answers from primary schools teachers who participated this school-year for the first time (N=972).

11.3.2 Sample Description

We provide here a description of the sample, 93,7% of which are women, apparently not far from the national value (96,4%) for primary school teachers. But this implies 6,3% are men, which is almost the double of the national value (3,6%): this appears to be a confirmation of the current situation where men are more attracted to computing than women.

This is the age distribution in the sample: up to 30: 8 (0.8%), 31 to 40: 133 (13.7%), 41 to 50: 415 (42.7%), 51 to 60: 374 (38.5%), 61 and more: 42 (4.3%), shown in figure 11.1.

Teaching seniority in the sample is the following: up to 2: 18 (1.9%), 3 to 5: 16 (1.6%), 6 to 10: 104 (10.7%), more than 10: 835 (85.8%), shown in figure 11.2.

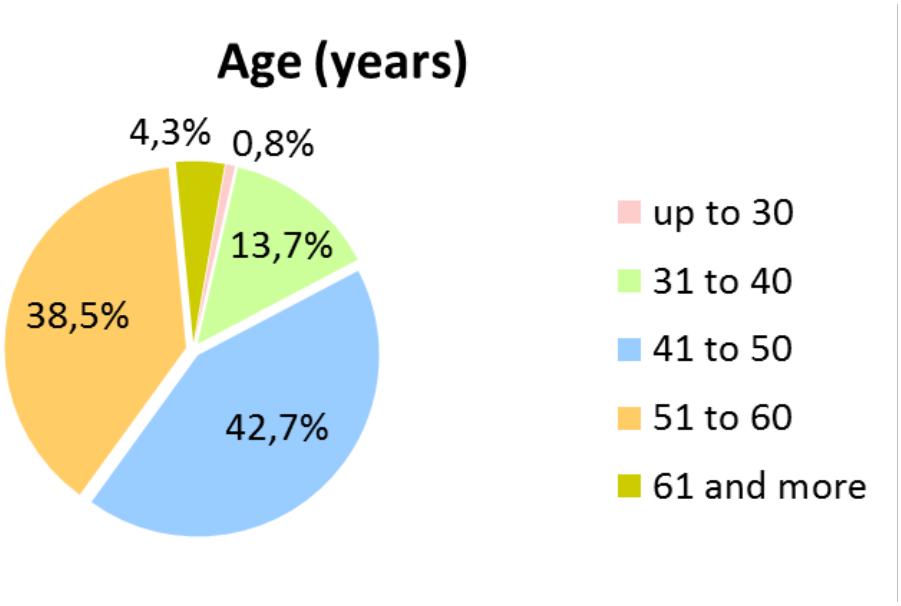


Figure 11.1: Age of teachers in years.

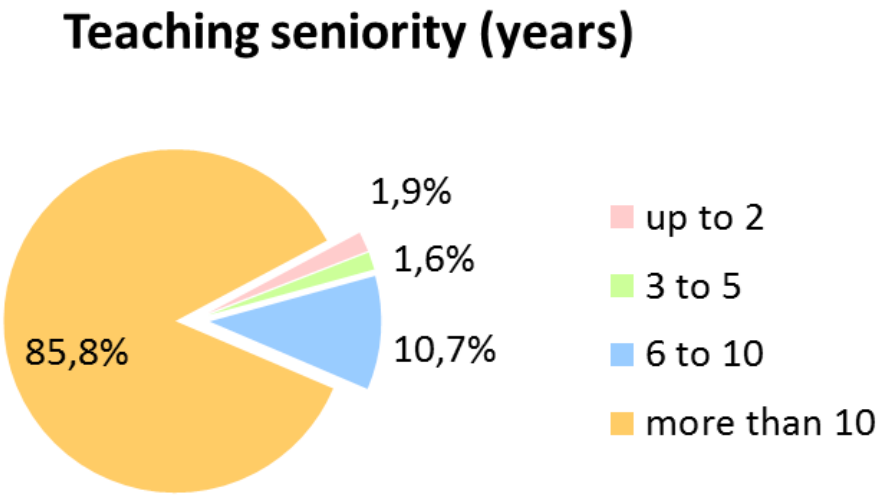


Figure 11.2: Teacher seniority in years.

Table 11.1: Areas of subjects taught by teachers.

Literary	49%
Informatics	10%
Scientific/ Technical	41%

Both of them show our sample is made, to a very large extent (>80%), by mature and experienced teachers. This grounds our findings on a reliable base of subjects, but on the other side indicates most probably they have not received any formal or structured training in informatics (see § 1.4.3).

See also in Table 11.1 the distribution of subject areas taught by teachers in our sample.

11.3.3 Procedures

We used a mixed methods approach, including both quantitative and qualitative analysis.

11.3.3.1 Quantitative Analysis of the First Part

We used standard descriptive statistical methods to analyze closed-ended answers (Q2 to Q5). More specifically, we computed the frequency distribution of these answers.

11.3.3.2 Qualitative Analysis of the First Part

Among the 972 answers, we filtered out those (116) that did not provide a definition and also those (77) that were completely out of scope (e.g.: they answered "interesting" or "useful").

We then proceeded to identify, by reading and discussing, the conceptual categories present in the remaining 779 definitions.

In a first phase each of the three researchers involved (among which there is the author of this dissertation) independently analyzed the definitions and proposed a set of conceptual categories to classify them. We used a mixed approach: some categories were defined "a priori", on the basis of literature (§ 2.2) and related work (§ 11.2), others were grounded on the definitions themselves. We then met to examine the proposed sets of categories and through discussion we agreed to a preliminary set.

We then manually assigned each answer to one or more category, if the statement either declared CT was of the same nature as the category or stated CT had relations to or was useful for the category.

For this process the set of answers was split in three, and each of us assigned answers in his/her set to one or more category. During this process proposals for modifications to categories emerged. Then we met again and jointly examined both these proposed modifications and assignments. Through discussion, we came to agree on the final set of 17 categories (described in subsection 11.5.1) and the final assignment of each definition to one or more category.

11.3.3.3 Measuring CT Knowledge

To be able to measure the *level* of teachers' knowledge about CT we used the following procedure. We assigned a weight (see discussion in subsection 11.5.1) to each category according to its relevance (in our view) for CT definition, in the light of the main definitions known in the literature (§ 2.2). Finally, the *level* of an answer was computed as the sum of weights of categories it is assigned to.

11.3.3.4 Quantitative Analysis of Second Part

We used standard descriptive statistical methods to analyze the frequencies of both actual and relevant answers Q6, Q7, and Q8. Moreover, we used the data and the model on CT definition of first part to analyze how the values provided by that model are distributed when restricted to answers to questions Q6, Q7, and Q8.

11.3.3.5 Qualitative Analysis of Second Part

We filtered out answers to Q6 that did not provide a definition (e.g. "innovative") and answers to Q8 that did not explained the difference (e.g. "coding is a discovery").

We then proceeded for each of the remaining *relevant* answers to identify, by reading and discussing, the conceptual categories to be used for their classification.

In a first phase each of us independently analyzed the definitions and proposed, for each question, a set of conceptual categories to classify them. We used a mixed approach: some categories were defined "a priori", on the basis of literature overview (§ 2.4), others were grounded on the definitions themselves.

Secondly, we jointly examined, for each question, the proposed sets of categories and through discussion we agreed to a preliminary set.

Subsequently, we manually assigned each answer to one or more categories, if the statement either declared the same nature as the category or stated being relative to or useful for the category. For this process the set of answers for each question was split between us, and we assigned answers in our own subset to one or more categories. During this process proposals for modifications to categories emerged.

Lastly, we jointly examined, for each question, both these proposed modifications and assignments. Through discussion, we came to agree on the final set of categories for each question (described in subsections 11.6.1.1 for Q6 and 11.6.3.1 for Q8) and the final assignment of each definition to one or more categories.

11.4 Technology and Preparation Perceptions

11.4.1 Q2 and Q3: Technology and Computational Thinking

The distribution of agreement with the two statements (Q2, Q3) investigating relations between computational thinking and technological devices are respectively shown in figures 11.3 and 11.4.

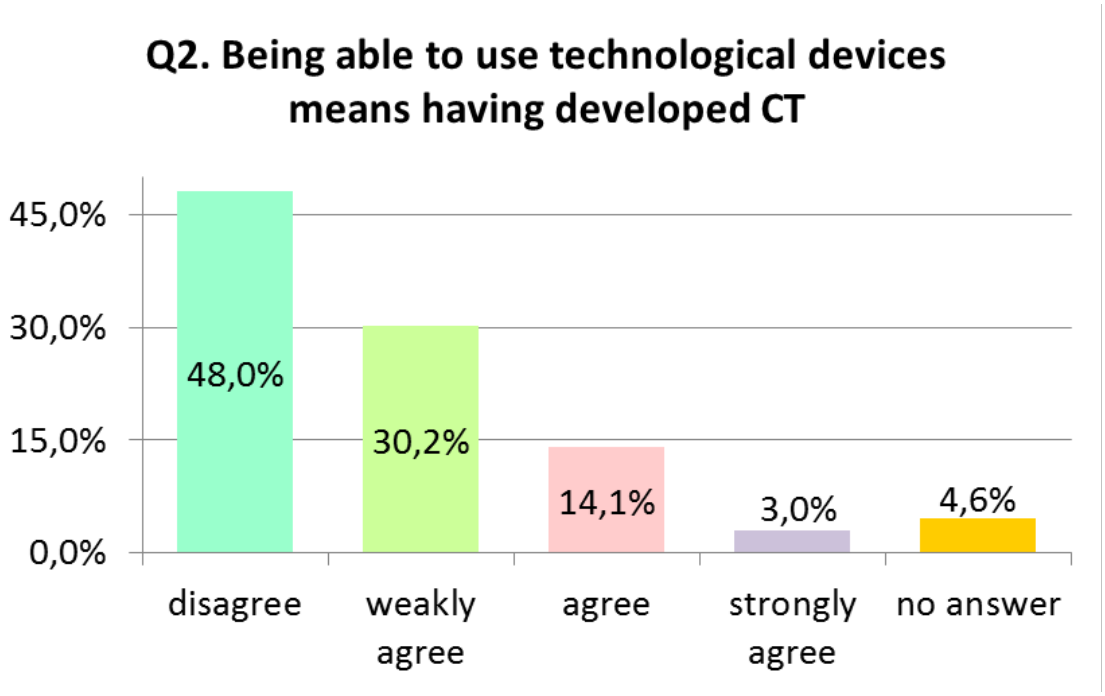


Figure 11.3: Technological devices and CT.

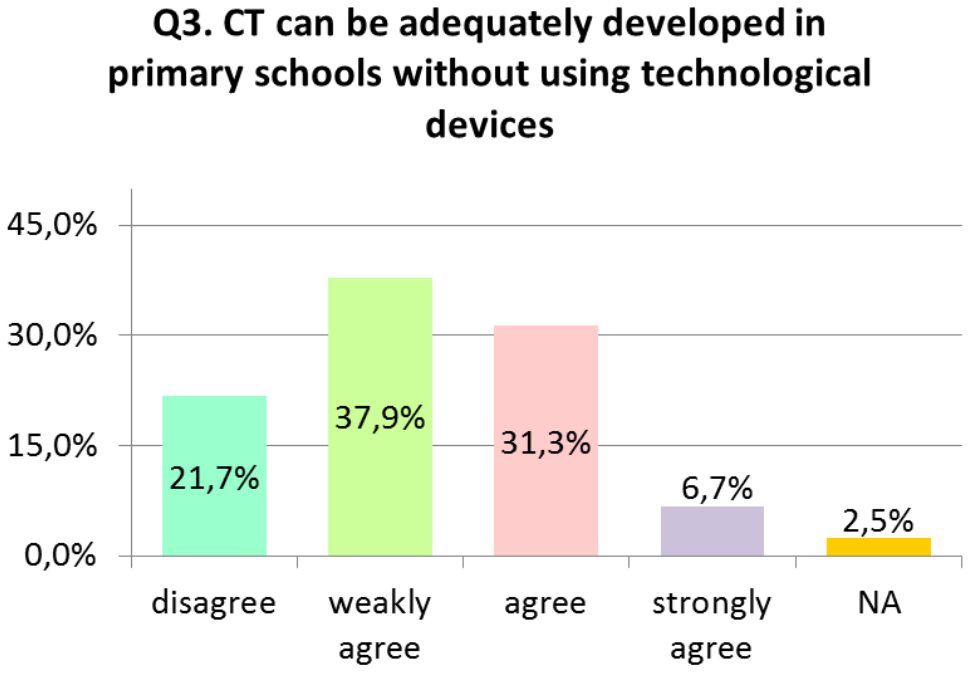


Figure 11.4: CT without technological devices.

It is positive that almost half of the teachers *disagree* with Q2, and just 17.1% *agree*s or *strongly agree*s with the statement. This shows Italian primary schools teachers have a sufficiently clear understanding that computing is not the same thing as using IT devices. This is supported by the qualitative analysis results discussed in section 11.3.3.2. We observe the results discussed in related studies [Yadav et al., 2011a;b; Bower and Falkner, 2015] appear to show a much higher level of misconceptions regarding CT in teachers but we note that those analysis were conducted on a different (smaller) sample (pre-service teachers) operating in a different culture (USA or Australia).

A positive insight is also given by answers to Q3. In fact, only less than a quarter (21.7%) of teachers thinks an adequate development of CT requires the use of technological devices, while 38.0% *agree*s or *strongly agree*s with Q3. We notice however that research on transfer (Chapter 4) and on CS Unplugged (Section 8.4.1) suggest that programming must be, at some point, introduced.

11.4.2 Q4 and Q5: Teachers' Preparation

Self-perception of teachers with respect to their level of preparation to develop CT competences in their students (Q4, Q5) is shown in figure 11.5.

It is apparent that a large majority does not feel adequately prepared. This is coherent with the fact that preparation of primary school teachers is not focused on specific disciplines but has a broad scope and there is not a specific training program in computing for school teachers of primary and lower secondary levels (see 1.4.3).

Figure 11.6 shows which initiatives teachers consider most important to improve their preparation (they could choose up to 3 items). Training is by far the most chosen one, which we feel is depending on the nature of our sample. This choice has also a rational support in the positive training effects noted in Yadav et al. [2011a;b]. A bit more worrying, in our view, is that slightly more than half of the teachers does not feel the need for *methodological guidelines* and just one quarter considers *learning objectives and teaching content* important to improve their preparation.

11.5 Q1: Teachers' Definition of CT

11.5.1 Categories

We now describe the 17 categories emerged from our analysis. We present them in four classes and indicate between parentheses the weight assigned to each category in a class for the purpose of the procedure described in 11.3.3.5.

- **Fundamental** (+2) - these categories express elements absolutely necessary in any definition of CT.

  `PSOL` Problem solving: action(s) or process(es) leading to solve a problem, to reach a goal, to face a complex situation.

  `MENT` Mental process or tool: a cognitive ability, a mental competence.

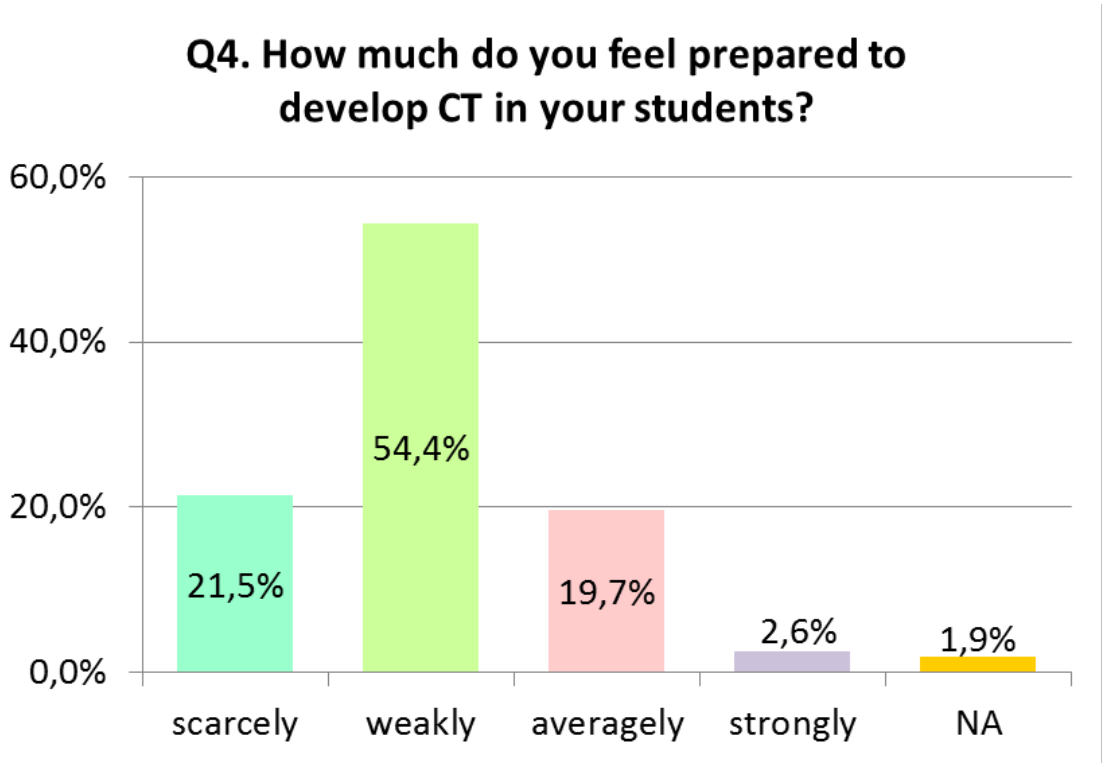


Figure 11.5: Teachers self-perception of their preparation.

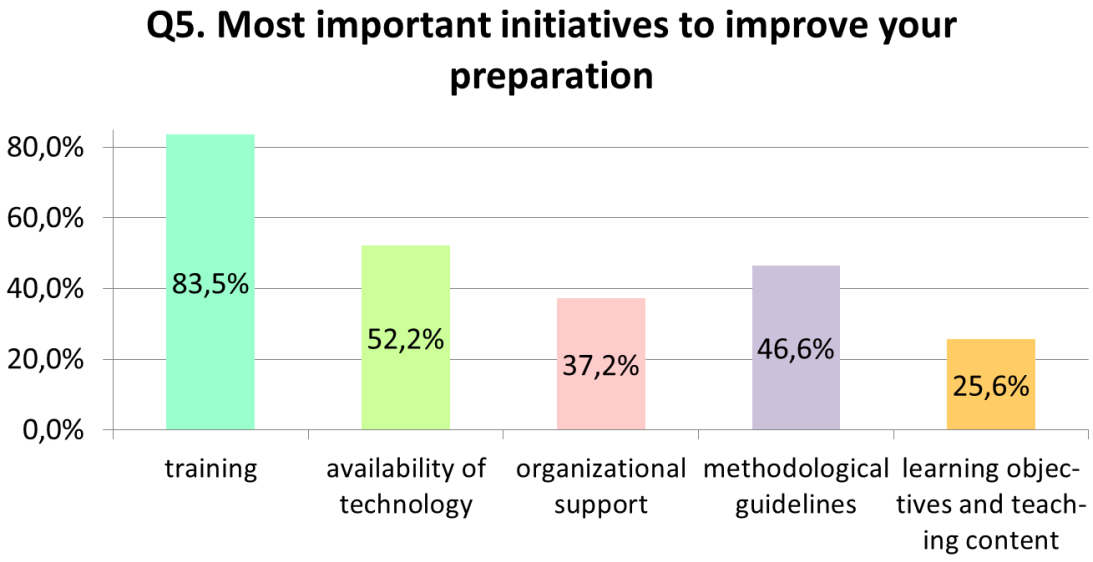


Figure 11.6: Most important initiative to improve teachers' preparation.

ALGO Algorithmic thinking: devising an algorithm to solve a problem; defining an effective method or strategy or plan; solving a problem by means of a sequence of elementary steps.

AUTO Giving instructions/automation: instructing some agent to solve a problem; providing a procedure to an information processing agent.

METH Using/learning informatics methods: the ability of using informatics concepts and methods; learning informatics.

- **Important** (+1) - these categories express elements that are important for a definition of CT but are not fundamental.

DECO Problem decomposition: splitting a complex problem in simpler subproblems to solve it more easily.

LOGI Logical thinking: logical or reasoning or analytical skills.

ABST Abstraction: focusing on common characteristic of general value; reusing a solution in other situations; devising a solution for a more general situation.

CODE Write programs: writing programs; coding.

- **Part-of** (0) - these categories express elements that are somehow present in definitions of CT reported in the literature; in some sense they are not necessary for a well-formed definition of CT.

MCOG Meta-cognition: reflecting about thinking or learning; learning to learn.

TRAN Transversal competence: e.g. fourth skill, transversal skill, life skill, useful in other fields, of general use, useful for teaching and learning.

CREA Creative thinking: being able to find creative or original solutions to problems; creativity.

UNIT Understanding information technology: understanding how information technology devices work; understanding science behind IT.

LANG Programming language: a language to communicate with IT devices.

ITER Iterative development: operating by means of successive refinements, possibly based on trial and error or testing and debugging.

- **Misleading** (-1) - these categories express elements whose presence in the definition of CT takes away from a correct understanding.

THPC "Think" like a computer: act mechanically like a machine, not behaving like a human.

UDEV Using IT: being able to use information technology devices and programs as an end-user.

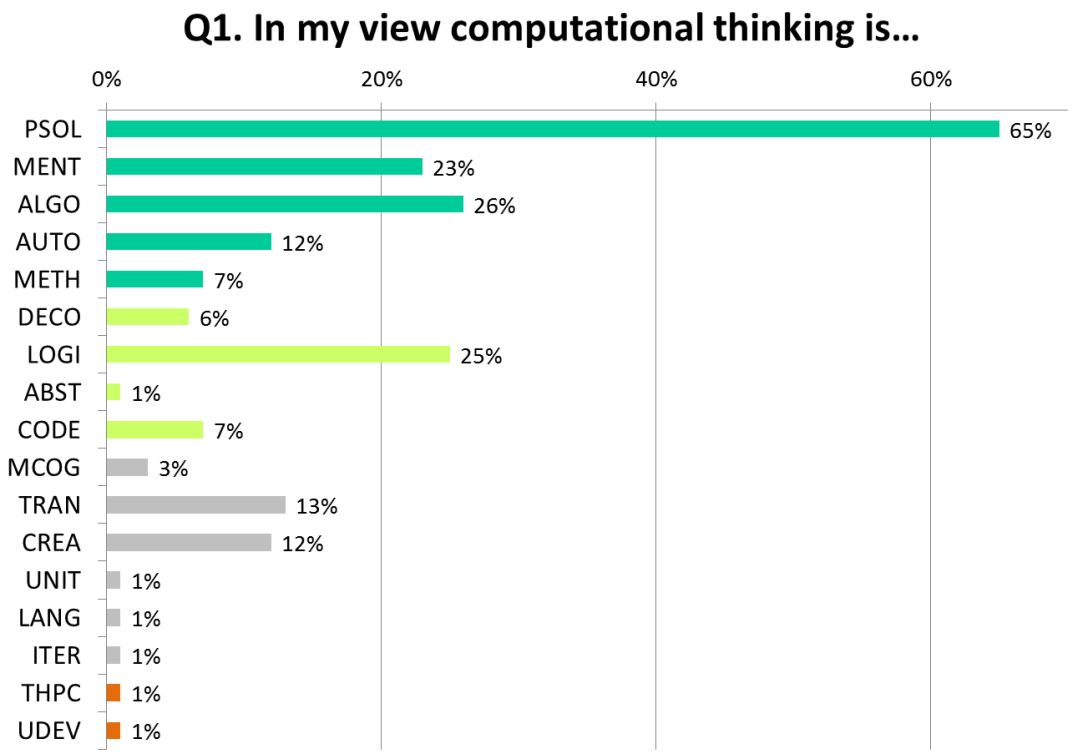


Figure 11.7: Frequency of each category in Q1.

11.5.2 Analysis of Category Distribution

The distribution of assignments of definitions to categories is shown in figure 11.7, where different colors code different classes (remember each definition was classified under one or more category).

To better understand the distribution and its analysis, note that "Programma il Futuro" website provides some introductory information[3], with a discussion on the role of computer science in the digital society and the importance of informatics as an autonomous scientific discipline, based on Informatics Europe [2013]. It also informally describes what CT is ("*Computational thinking is a mental process for problem-solving with distinctive techniques and general intellectual practices*")[4]. This may explain why two thirds of the answers identified *problem solving* as an element of the definition of CT.

We note that some categories have a surprisingly high frequency relatively to their importance (in our view) for the definition of CT: *logical thinking*, *transversal competence*, and *creative thinking*. A possible motivation is that a Google query in Italian about computational thinking returns these terms in the first few results.

Also, it is somewhat surprising the low frequency of use of *abstraction* to characterize

[3] https://programmailfuturo.it/progetto/perche-partecipare
[4] https://programmailfuturo.it/progetto/cose-il-pensiero-computazionale

CT, given the very strong stance taken by Wing in respect to it. We think the conceptual difficulty inherent with the role of abstraction in informatics may explain this outcome.

11.5.3 Analysis of Answer Values Distribution

11.5.3.1 Approach

Since all definitions (with their constitutive elements) of CT considered in section 2.2, if classified and evaluated with our procedure in 11.3.3.3, have a value of at least 8, we decided to use this as the threshold to identify the class of "good definitions". We also set the "acceptable definition" threshold at 6. In fact, to reach 6, an answer must have been labeled with categories defined in subsection 11.5.1 so that it falls within one of the following cases:

(c1) at least 3 fundamental

(c2) 2 fundamental and at least 2 important

(c3) 1 fundamental and 4 important

In other words, there is no way for an answer to be evaluated as an "acceptable definition" if it does not have at least 1 fundamental. But 1 or 2 fundamental alone are not enough, if they are not accompanied by a large enough number of important.

We consider all definitions whose value is 5 or less as misconceptions.

11.5.3.2 Outcome

Our procedure evaluates just 8 of the 779 answers as "good definitions". The number of those being acceptable but not good are 76, resulting in a total of just about 10.8% of all answers being at least acceptable. This result appears to be correlated with the feeling of a weak preparation reported in figure 11.5.

Also, all of not acceptable answers with a value of 5 and 96% of those with a value of 4 have at least 2 fundamental. This leads to a comforting 43.4% of answers that features the presence of at least two fundamental components for a CT definition.

The 695 not acceptable answers (i.e., the misconceptions) are roughly evenly split among those with a value at least 3, and those with a value less than 3: see in figure 11.8 the overall distribution.

Moreover, we investigated the frequency with which each couple of categories appeared in the definition. We report in table 11.2 the 27 most frequent couples with at least 2% frequency. This table therefore shows the frequency with which (at least) both categories have labeled answers, that is their frequency of co-occurrence. Row and column headings appear in the same order as in subsection 11.5.1 and, to make the table more compact, not all categories are listed.

Note that PSOL plays a leading role, which is understandable given two thirds of definitions have received its label. A positive element is the relatively high frequency of co-occurrence of PSOL with ALGO (22%): this can be interpreted as an evidence that that about 11% of answers (22%-10.8%), even if not acceptable, are characterized by a sound (even if incomplete) description of computing.

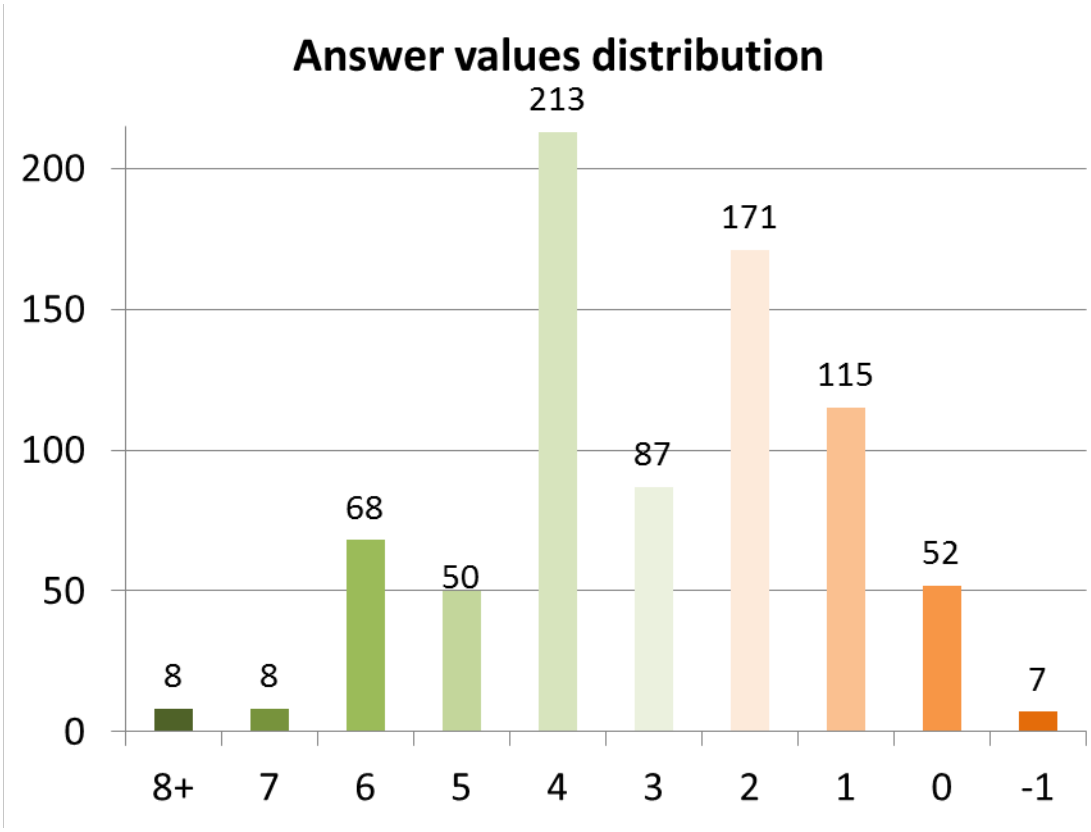


Figure 11.8: Answer values distribution.

Table 11.2: Most frequent (%) couples of categories.

	MENT	ALGO	AUTO	METH	DECO	LOGI	CODE	TRAN	CREA
PSOL	17	22	8	5	5	10	2	6	8
ALGO	4				2	4		2	
AUTO	2	2				2	2		
METH	4							2	
LOGI	2				2			2	
TRAN	5								
CREA	2	3				4		2	

11.5.4 Conceptions and Misconceptions Regarding CT

Examining all the answers that are at least acceptable we observe just 29 distinct sets. Table 11.4 on next page shows the count, the value according to our procedure in 11.3.3.3, and the constituent categories of each set.

Table 11.3: Distinct sets and counts of not acceptable answers

Value	Count	Labels
...		
4	88	PSOL, ALGO
4	51	PSOL, MENT
4	28	PSOL, AUTO
...		
3	24	PSOL, LOGI
3	11	PSOL, DECO
...		
2	80	PSOL
2	19	PSOL, CREA
2	11	MENT
...		
1	67	LOGI
1	13	CODE
1	11	LOGI, CREA
...		
0	26	TRAN
...		

There is no example of a set belonging to case (c3), see 11.5.3.1, and just 4 lines of the table show sets belonging to case (c2), explicitly indicated in the table, meaning the overwhelming majority of acceptable answers belongs to case (c1).

Moreover, three different sets have a high count (marked with a $*$ in the "Case" column): {PSOL, MENT, METH}, {PSOL, MENT, ALGO}, and {PSOL, MENT, METH TRAN}. We think this is a positive result since these answers are all instances of case (c1), even if many examples in these sets are clearly molded after the information provided in "Programma il Futuro" website.

The most frequent not acceptable answers are shown in table 11.3.

A large number of misconceptions is characterized either by PSOL alone or by its coupling with exactly one of these categories: MENT, LOGI, DECO, CREA (first 7 lines of the table). In all these cases a very partial view of informatics emerges, given the absence of categories describing the *information-processing agent*. A similar situation happens for MENT and LOGI (next 2 lines). This reinforces our concerns that considering CT as a subject somewhat distinct from computing may give raise to misconceptions.

Another misconception is shown by the relatively high count of TRAN alone (last line), which shows the evidence of a view of CT as an instrumental discipline, not important in itself. This is possibly deriving from attempts to convince teacher of the importance of CT by focusing mainly on its value for other disciplines and as a general learning tool.

11.6 Conceptions on Coding

11.6.1 Q6 - Coding is...

A definition was present in 88% (854) of all 972 answers. Among the 118 ones which did not provide it, 50.8% (60) did not answer to Q7 either, 24.6% (29) answered negatively with respect to the difference between *coding* and *writing programs* while a same (by chance) 24.6% (29) answered positively (but only 2 answers contained an explanation). Among the 854 provided definitions, 7% (56) of them were not relevant (e.g., "coding is innovative").

11.6.1.1 Categories

The qualitative analysis of the remaining 798 (=854-56) relevant answers to Q6 using the procedure described in 11.3.3.5 resulted in 10 categories. We grouped them in two classes, according to whether they were somewhat related to *writing programs* or not.

- **Related**: All categories here somehow "speak" about writing programs, either in a full (PROG) or simplified (SIMP) way, or are concerned with writing algorithms (or lists of instructions) making reference to some information processing agent able to execute them mechanically (PROC).

  PROC *Specifying processes*: devising an algorithm to solve a problem; providing a list of instructions to solve a problem; making an information processing agent execute a sequence of elementary steps

  PROG *Writing programs*: using programming languages

  SIMP *Simplified programming*: programming with simplified environments/ languages (e.g.: visually, blockly); learning the basics of programming

- **Unrelated**: Categories in this class are not directly concerned to writing programs in some form.

  ACTI *Being active towards information technology*: creating computational artifacts instead of simply using them; being able to find creative or original solutions to problems

  COLE *Cognition and learning*: reflecting about thinking or learning; program to learn; learning to learn; develop/ improve cognitive abilities; a method/ approach to teaching/learning

  DECT *Developing computational thinking*: a way to teach/ develop/ apply CT

  ENGA *Engagement*: doing playful/ funny/ attractive/ interesting/ inspiring activities

Table 11.4: Distinct sets and counts of acceptable answers

Value	Count	Case	Labels										
12	1		PSOL	MENT	ALGO	AUTO	METH	DECO	LOGI				
11	1		PSOL	MENT	ALGO		METH	DECO	LOGI	ABST		CREA	
9	1		PSOL	MENT	ALGO		METH			ABST			
8	3		PSOL	MENT	ALGO	AUTO							
8	1		PSOL	MENT	ALGO		METH				TRAN	CREA	
8	1		PSOL	MENT	ALGO		METH						
7	2		PSOL	MENT	ALGO			DECO					
7	2		PSOL		ALGO	AUTO			LOGI				
7	1		PSOL	MENT			METH		LOGI		TRAN	CREA	
7	1		PSOL	MENT	ALGO					ABST	TRAN		
7	1		PSOL		ALGO		METH		LOGI				
7	1		PSOL		ALGO	AUTO				ABST			
6	17	*	PSOL	MENT			METH						
6	11	*	PSOL	MENT	ALGO								
6	10	*	PSOL	MENT			METH				TRAN		
6	7		PSOL		ALGO	AUTO							
6	6		PSOL	MENT		AUTO							
6	2	c2	PSOL		ALGO			DECO	LOGI			CREA	
6	2	c2	PSOL		ALGO			DECO	LOGI				
6	2		PSOL	MENT	ALGO						TRAN	CREA	
6	2		PSOL	MENT	ALGO						TRAN		
6	2		PSOL	MENT	ALGO							CREA	
6	1	c2	PSOL			AUTO		DECO	LOGI				
6	1		PSOL	MENT		AUTO					TRAN		
6	1		PSOL	MENT		AUTO							ITER
6	1		PSOL		ALGO	AUTO					TRAN		ITER
6	1	c2	PSOL		ALGO				LOGI	ABST			
6	1		PSOL		ALGO		METH						
6	1		PSOL			AUTO	METH						

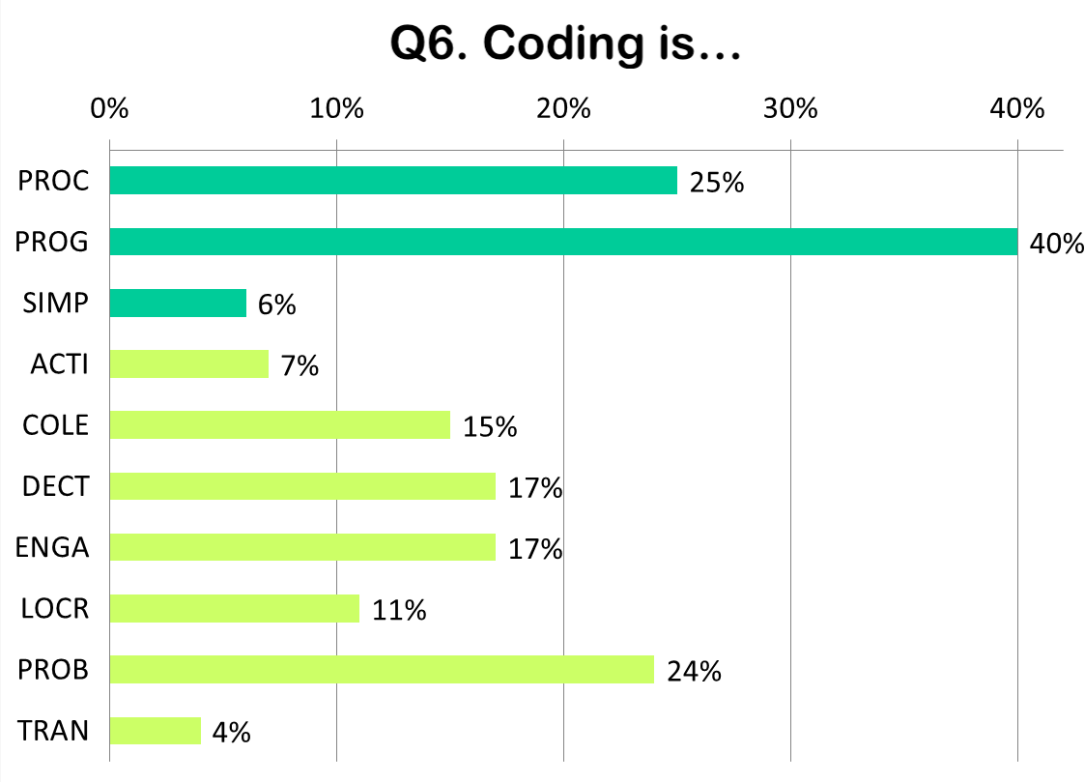


Figure 11.9: Frequency of each category in Q6.

LOCR *Logical/critical thinking*: logical or reasoning or analytical skills; applying/developing critical thinking

PROB *Solving problems*: plan(s), design(s), action(s) or process(es) leading to solve a problem, to reach a goal, to face a complex situation (including splitting a complex problem in simpler subproblems to solve it more easily)

TRAN *Transversal competence*: e.g. fourth skill, transversal skill, life skill, useful in other fields, of general use

11.6.1.2 Analysis of Category Distribution

The distribution of categories for the 798 relevant answers to this question is shown in figure 11.9.

Category PROG, which directly relates *coding* to *programming*, is understandably the most frequent one, but appears in only 4 out of 10 relevant answers (323/798). If only the 456 teachers answering also "no" to Q7 are analyzed, this percentage slightly increases to 43% (194/456), a slightly positive sign that those teachers correctly relating "coding" and "writing programs" (i.e., the "no" answers to Q7) were also better able to describe coding in terms of programming.

On the other side, by aggregating the answers in the *related* class (i.e., PROG, PROC, and SIMP - remember each answer can receive more than one label) we obtain that 59% (469) of answers relevant for Q6 use an expression somewhat related to *programming*. In the light of the characteristics of our sample (see 11.3.2), the fact that 6 out of 10 teachers were somewhat able to identify a correct relation between *coding* (which for large part of their professional career most probably they never heard about) and *programming* is certainly positive.

Also note that, given each relevant answer was assigned to one or more categories, there is a 29% (232) of answers falling both in the *related* and *unrelated* classes. On the other side, there was a 30% (237) of answers belonging to at least one of the *related* categories and none of the *unrelated* ones and a 41% (329) of answers belonging to one of the *unrelated* ones and none of the *related* ones.

The third most frequent category is PROB, that with a 24% (190) is very close to the 25% (197) of the second one, PROC, confirming the trend emerged for CT (11.5.2), that in Italian schools CS education is often considered as a general instrument for problem solving. The strict relation between CT and programming is confirmed by the 17% (138) seeing coding as a way to teach/ develop CT (DECT).

It is interesting to note that 17% (138) of teachers highlights the *engagement* value of coding (ENGA) and 15% (117) sees it as an aid for teaching or developing cognitive abilities (COLE). We think these are important elements to ensure a diffusion of computing education in schools, although one has to pay attention they do not overshadow its core elements. The same reasoning applies to LOCR (11%, 87) and TRAN (4%, 31).

Only a 7% (54) of teachers has remarked the importance of coding to become active towards Information Technologies (ACTI), which is anyway positive given the question was not investigating the role/ purpose of *coding*.

11.6.1.3 Relationship with CT Definition

Among the 798 relevant answers to Q6 there were 743 who also provided an admissible CT definition.

We show in Table 11.5 how these 743 definitions are distributed according to two subsets: one made up by all 458 answers to Q6 using terms somewhat *related* to "writing programs" and the other one made up by the 285 remaining answers. The average value of the CT definition evaluation model for all *related* teachers is 3.37, while for all the remaining ones is 2.62, showing a positive correlation between understanding CT and being able to properly define "coding".

11.6.2 Q7 - Is coding different from writing programs?

An answer was provided to Q7 by 78% (758) of the 972 teachers in the sample and 60% (456) of them answered "no" and 40% (302) answered "yes". Among the 214 ones which did not provide an answer, 28% (60) did not answer to Q6 either.

Table 11.5: Distribution of CT values of relevant Q6 answers.

value	-1	0	1	2	3	4	5	6	7	8	9	10	11	12
Rel.	1	16	40	105	57	149	32	47	6	3	1	0	1	0
Remain.	5	27	65	58	28	60	16	21	2	2	0	0	0	1

11.6.2.1 Relationship with CT Definition

We used the CT definition evaluation model and the related set of data in 11.5 to analyze the distributions of values provided by such model when restricted to the two significant subsets of possible answers to our question Q7 (namely, "yes" or "no").

The sample of 972 answers contained 779 answers with admissible CT definitions. We analyzed the admissible ones and found 396 "no" and 246 "yes" answers to Q7, while 137 did not answer at all. The percentage of *acceptable* CT definitions[5] is slightly higher for the admissible CT answers who also answered "no" to Q7 (12%, 46/396) than for those who answered "yes" (10%, 25/246).

This shows that teachers having correctly identified that there is no difference between *coding* and *writing programs* have performed slightly better, for what regards the definition of CT, than those who think there is a difference. This is confirmed also by comparing the average value for acceptable CT definitions in the two subsets of teacher having answered "no" (avg $= 6.33$) and "yes" (6.12).

11.6.3 Q8 - The difference between coding and writing programs is...

Among the 302 answers to Q7 incorrectly stating *coding* and *writing programs* are different, only 53% (159) explained why by answering Q8: 25 of these were not relevant, while the qualitative analysis of the remaining 134 ones is provided in the following section.

11.6.3.1 Categories

The analysis of the 134 relevant answers to Q8 using the procedure described in 11.3.3.5 resulted in the 11 categories described below. We grouped them in three classes according to how they described the difference between "coding" (C, in the following description) and "writing programs" (P). Some descriptions are tolerable while others are unacceptable. A few are completely out-of-scope.

- **Tolerable**: in this class we have categories expressing admissible relations, given the wide variety of ways in which the two terms are used in both literature and profession.

`COMP` - C is a part of P

`EASY` - C is a simplified P

[5] Remember that a value of at least 6 characterizes an "acceptable" CT definition

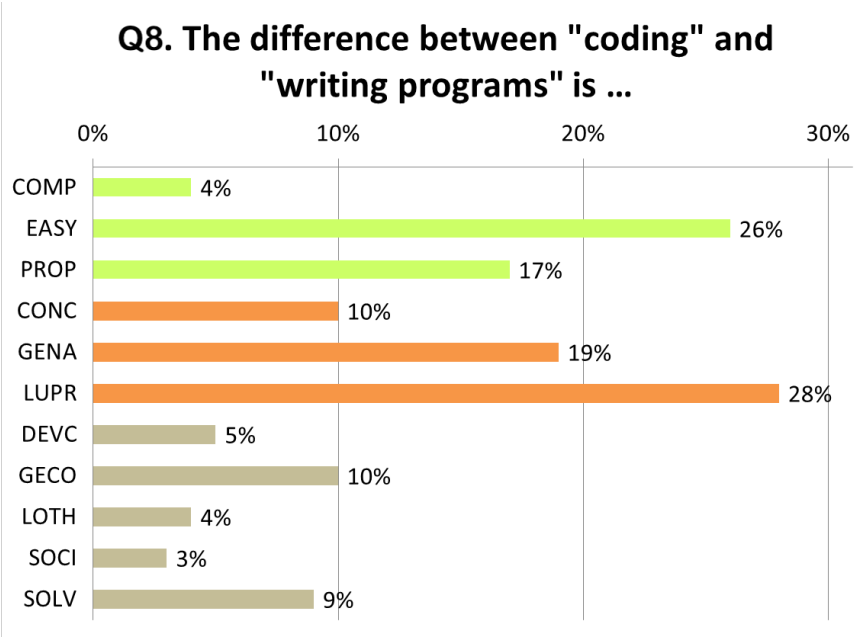


Figure 11.10: Frequency of each category in Q8.

PROP – C is preparatory to P

- **Unacceptable**: categories in this class refer to wrong ways of describing relations between C and P.

CONC – C is the conceptual part of P

GENA – C is more general/ abstract than P

LUPR – C is for playing/ learning, P is for working

- **Out of scope**: here we have categories which do not really address the difference but simply refer to characteristics of C.

DEVC – C is a means to develop computational thinking

GECO – C is a general competence

LOTH – C is a means to learn other subjects

SOCI – C has a social value

SOLV – C is problem solving

11.6.3.2 Analysis of Category Distribution

The distribution of categories is shown in figure 11.10.

In 43% (58) of relevant answers there were elements characterizing *coding* as simpler than (EASY, occurring in 35 answers, 26%) or preparatory to (PROP, 23, 17%) or part of (COMP, 5, 4%) *programming*: we collectively denote these answers as *tolerable*. In 54% (72) of cases there were elements expressing (at least one) actual (and *unacceptable*) misunderstanding of the relation between *coding* and *programming*. There were 17 answers with both *tolerable* terms and *unacceptable* ones.

We classified these misunderstandings in three *unacceptable* categories: both CONC (characterizing 13 relevant answers, 10%) and GENA (25, 19%) reverse the position, expressed in the literature (see 2.4), that considers *coding* as a narrower concept than *programming* in the software production process. The former by assigning it a conceptual role CT is concerned with, the latter by ascribing to *coding* a scope wider than the mere act of writing programs.

Answers classified as LUPR (28%, 38) have elements characterizing *coding* as just a "toy" activity distinct from "the real thing", that is programming in a professional context. Clearly, this misunderstanding goes in the opposite direction as the two previous ones, and none of the 134 relevant answers was self-contradictory by expressing both LUPR and (CONC or GENA).

11.6.3.3 Relationship with CT definition

Among the 779 admissible CT definitions of Section 11.5 there were 123 who also answered to Q8, and none of these definitions has a value greater than 6 according to their model. Also, the percentage of those receiving a value at least 6 (acceptable definition) is lower in this subset of teachers (7%, 9/123) than in the overall set (11%, 84/779) and the percentage of unacceptable (<6) definitions in this subset (93%, 114/123) is higher than in the overall set (89%, 697/779).

The fact that teachers having tried to characterize a difference between *coding* and *writing programs* were not able to provide an acceptable CT definition shows an agreement between this research and the evaluation provided by the model in 11.5.

Figure 11.11 compares the overall distribution of values from the model in 11.5 to the distribution of values restricted to those 159 teachers who answered Q8.

11.6.3.4 Reconsidering the Relation Between Coding and Writing Programs

Figure 11.12 shows the Venn diagram of the classification of answers to Q8 according to the 3 classes grouping the 11 categories. A minority of the 58 relevant answers classified in the *tolerable* class were also classified in the *unacceptable* one, but there were 41 whose classification did not have *unacceptable* categories.

These are therefore teachers considering *coding* as distinct from *writing programs*, but whose answers can be aggregated with the 456 negative answers to Q7 and removed from the positive ones. We thus obtain a 66% (497) of the 758 teachers who answered Q7 having an acceptably correct view of the relation between *coding* and *writing programs*.

Finally, we have also analyzed the subset of teachers whose answers to Q6 featured both a classification as (PROG or SIMP) and as PROC: there are 71 of them. In this subset of *strongly related* answers (9% of the 798 relevant answers for Q6) there are 51 (72% of the subset) who were able to correctly relate *coding* to *programming* (in the enlarged sense described

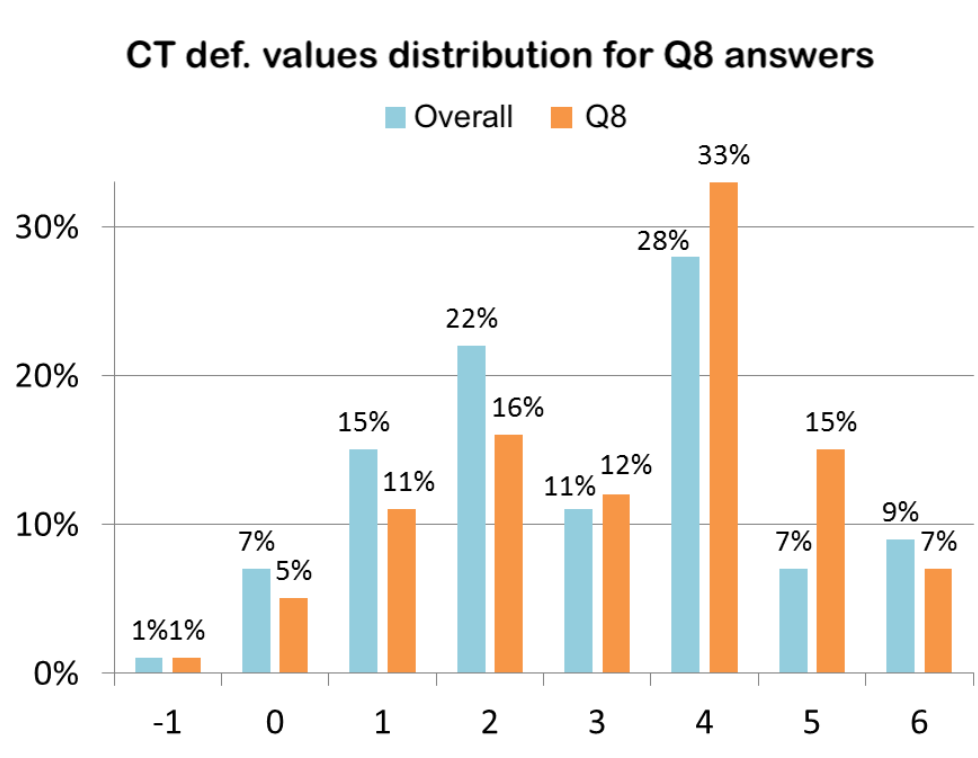


Figure 11.11: Distribution of CT value for Q8 answers.

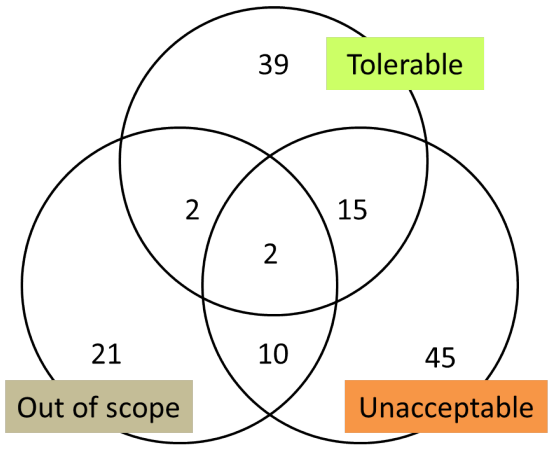


Figure 11.12: Distribution of Q8 answers among classes.

Table 11.6: Distribution of CT values for a subset of Q8 answers.

value	-1	0	1	2	3	4	5	6
Tolerable-only	0	1	2	8	7	12	5	4
Unacceptable-only	0	3	5	7	4	18	7	4

in 11.6.3.2) while only 1 of them described an *unacceptable* difference between *coding* and *writing programs*.

11.6.3.5 Relationship with CT Definition

Let us now define as *tolerable-only* answers the 41 ones classified with at least one *tolerable* category and none *unacceptable*, and as *unacceptable-only* the 55 ones with at least one *unacceptable* and none *tolerable* (see again figure 11.12).

Table 11.6 shows the values of CT definitions provided from the model in 11.5 for these two subsets. Remember that no teacher who answered Q8 received a value greater than 6 in the CT definition evaluation. Note that there are fewer answers in *tolerable-only* subset than in *unacceptable-only* (39 vs 48), and teachers in the *tolerable-only* subset have a slightly higher average value (3.49 vs 3.38) for the CT definition evaluation model.

11.6.4 Joint Distribution of Q6 and Q8

All the 134 teachers answering Q8 answered also to Q6. We present in Table 11.7 the joint distribution of all answers to Q6 and those answers to Q8 classified as *tolerable-only* or *unacceptable-only*.

Table 11.7: Joint distribution of answers to Q6 and Q8.

	Tolerable-only	Unacceptable-only	SUM
Related	28	22	50
Remaining	13	33	46
SUM	41	55	96

In Tables 11.8 and 11.9 we present the two respective marginal distributions.

Table 11.8 shows that both a majority (68%) of relevant answers classified as *tolerable-only* with respect to Q8 belong to *related* with respect to Q6 and a majority (60%) of relevant answers classified as *unacceptable-only* with respect to Q8 belong to *remaining* answers in Q6 (i.e. the subset of Q6 answers that has not been classified in any of the *related* categories). This indicates a positive correlation between the capability of describing what coding is (Q6) and the capability of providing a description of the difference between coding and programming (Q8).

Table 11.8: Marginal distribution of Q8 answers with respect to Q6.

	Tolerable-only	Unacceptable-only
Related	**68%**	40%
Remaining	32%	**60%**
SUM	100%	100%

Table 11.9: Marginal distribution of Q6 answers with respect to to Q8.

	Tolerable-only	Unacceptable-only	SUM
Related	**56%**	44%	100%
Remaining	28%	**72%**	100%

This is confirmed also by Table 11.9, showing that both a majority (56%) of relevant answers classified as *related* with respect to Q6 belong to *tolerable-only* with respect to Q8 and that a majority (72%) of the *remaining* relevant answers to Q6 belong to *unacceptable-only* with respect to Q8.

11.7 Conclusions and Further Work

Outcome of our work shows the vast majority of Italian primary school teachers has not a sound and complete conception about CT (RQ1).

This negative finding is somewhat balanced by the evidence regarding teachers in relation to information technology (IT). In fact, it is sufficiently clear to them that (1) computer science and the use of IT are two distinct fields, and (2) IT devices are not absolutely needed to develop CT competences in students (RQ2).

Thirdly, teachers feel themselves not enough prepared to develop CT competences in their students and identify in specific training the most important initiative (RQ3).

What we reported in this chapter is only a first analysis of the situation in Italy regarding CT in schools in relation to "Programma il Futuro".

Moreover, we analyzed Italian primary school teachers' ideas about coding and its relationship with programming. Regarding RQ4, we found that only 4 answers out of 10 directly mentioned *programming* when defining *coding*. On the other hand, if we consider also answers mentioning simplified programming environments/languages or the act of designing algorithms or giving instructions to an executing agent in the definition of *coding*, the number of good answers grows to a more comforting 6 out of 10. Answers highlighted also side aspects of coding, often overlapping with elements more rightly belonging to CT.

For what concerns RQ5, when directly asked if there is a difference between *coding* and *write programs*, 60.2% of them answered *no*. Another 5.4%, even if answered *yes*, gave a

completely tolerable explanation in the light of the variety of ways the term *coding* is used in different contexts, resulting in an overall 2 out of 3 teachers expressing an acceptably correct relation between coding and programming. The others giving an explanation (half of those answering *yes*) listed some characteristics of *coding* without really explaining the difference or used wrong (and conflicting) motivations: coding is the conceptual part of programming, or more general and abstract than programming, or just a toy while programming only for professionals.

We also compared our findings with ideas our sample had about CT. We found that teachers having acceptable ideas about it performed slightly better in: (i) describing coding with programming-related terms, (ii) correctly identifying *coding* with *writing programs*.

Finally, when comparing *coding* definitions and explanations of differences between *coding* and *writing programs*, we found that 68% of those who provided a programming-related definition managed to provide also a completely tolerable explanation of the difference. Dually, the vast majority of those who failed to relate coding to programming activities in its definition also provided unacceptable motivations for the difference.

Despite being limited to Italy, our study - showing that teachers have not a clear picture of CT and on the relations between coding and programming - can be representative of similar situations in K-12 education of many developed countries. We plan to extend our analysis to teachers of other school levels and to compare these results with those of teachers with more experience in CS teaching. It would also be interesting to carry out surveys in other countries to obtain a wider picture of the relations between misconceptions regarding CS related terms and teacher training.

The most probable cause for the misconceptions revealed by this study is the fact that teachers have not been appropriately trained in CS and its teaching methods. The importance of teacher training has already been identified in other reports [e.g., The Royal Society, 2017; The Committee on European Computing Education (CECE), 2017] as a key factor for a successful uptake of CS education in schools. These results support worries about the fact that focusing only on a specific activity/ tool of CS (i.e., on programming) can be harmful, especially if referring to it with a "buzzword" like *coding*, which takes on conflicting meanings. In fact, our results show that such misconceptions have a high correlation with the presence of reductive or wrong ideas about CS. On the other hand, having an appropriate understanding of what coding and programming are is an important requirement for teachers to be able to provide good CS education in schools. This research has shed some light on the fact that lack of proper training joined with confusion in terminology spread by media originated dangerous misconceptions, which may harm effectiveness of CS education actions. We therefore recommend, when speaking about CS education, to stress the importance of CS scientific principles and methods. As discussed in Chapter 6, it has to be clearly stated that CS (and not CT or coding) is the scientific discipline to be taught at all school levels [Lodi et al., 2017; Nardelli, 2019], both because it is the science underpinning the development of the current digital society and because it provides conceptual methods contributing to a better understanding of other disciplines. This has to be done at a communication level when presenting and discussing CS school education initiatives, at the organizational level of CS school curricula specification and in the context of teacher training in CS.

Part IV

Implicit Theories

Chapter 12

Studying Computer Science Does Not Automatically Foster a Growth Mindset[1]

As seen in Chapter 1, many arguments are used to advocate for the introduction of CS in K-12 education. At the same time, as described in Chapter 5, mindset theory is also becoming very popular among educators and researchers.

Some claims stating that studying CS can foster a growth mindset (GM) have emerged. However, educational research shows that the transfer of competences is hard (see Chapter 4). Moreover, very little research has been conducted on the relationship between GM and CS learning, with conflicting results.

In this Chapter, we will report a research study in which we measured some indicators (e.g., mindset, computer science mindset) at the beginning and the end of a high school year in five different classes: three CS oriented, one Chemistry oriented, and one Transportation&Logistics oriented. In one of the CS oriented classes, we did a very brief mindset intervention.

At the end of the school year, none of the classes showed a statistically significant change in their mindset. Interestingly, non-CS oriented classes showed a significant decrease in their *CS growth mindset*.

In the intervention class, students suggested, to stimulate a growth mindset, the need for activities that are more creative, engaging, and related to the real world and their interests.

We will report the context, methodology, and data analysis of the study. We will discuss the results in light of previous research and point out the implications of these results and future perspectives.

12.1 Introduction

Some claims stating that studying CS/CT/programming/"coding" can foster a growth mindset have emerged: partially because of the popularity in educational contexts, partly

[1]This chapter is based on material published in [Lodi, 2019].

because some characteristics of Computer Science - and in particular of programming (e.g., collaboration, iterative development, continually finding and correcting errors) - intuitively fit well in the mindset theory. The thesis that learning computing can foster a growth mindset is exemplified by some quotes from online educational blogs.

> Not only can the process of learning to code be used to help develop a growth mindset, there are aspects of coding that help reinforce this mindset. [Venkataswamy, 2016]

> [...] its contribution to a growth mindset, is one of the reasons why debugging is such an important part of computing education.
>
> [Berry, 2015]

> But did you know that by learning to code, you can simultaneously grow your growth mindset in the process? [Villanueva, 2018b][2]

> Over three years of helping kids teach themselves computer programming, I have decided that youth coding programs are the best way to instill a growth mindset in another human being.
>
> [Smith, 2016]

> Posing challenging tasks such as basic coding even at the elementary level pays great dividends and fosters a Growth Mindset among even the youngest students. [Hallett, 2016]

Other CS Ed. researchers seem to acknowledge the problem. For example, Lewis states, talking about growth mindset and grit, that "*programming has been speculated to be uniquely qualified to help normalize failure and thus encourage productive learning strategies*" [Lewis, 2017, p. 18]. As seen in Chapter 4, and recognized by Lewis herself, research in education tells us that transfer is difficult and unlikely to happen, especially between knowledge domains far from one another, and especially when treating domain-general / higher-order thinking skills.

To better understand the relationship between studying or not studying Computer Science and mindset (and other related constructs, especially CS mindset - see Section 5.2), we decided to conduct a study in a high school with different tracks. One track is Computer Science oriented, while others are oriented to different technical skills (e.g., Chemistry, Logistics and Transportation). With our experiment, we tried to address the following research questions:

RQ1 Does studying or not studying Computer Science in high school automatically influence students' mindset, Computer Science mindset, and other related constructs?

RQ2 Can a small explicit intervention alter students' mindset?

RQ3 After being taught about mindset theory, what are students' recommendations to foster it?

[2] Another blog post on Mindsetworks also seems to acknowledge the opposite direction of the implication: Having a Growth Mindset helps learning to code [Villanueva, 2018a]

Table 12.1: Descriptive statistics of the sample

Class	N	Males	Females	Prev. exp
INF1	16	16	0	7
INF2	14	11	3	1
INF3	7	7	0	7
TRAN	6	6	0	3
CHEM	23	16	7	12
ALL	*66*	*56*	*10*	*30*

12.2 The Study

12.2.1 Participants

We conducted the experiment in four classes (third year, 16-year-old students) of a large technological institute (in brackets we give a short code we will use to identify the classes): 2 classes of "IT and telecommunications" (INF1, INF2); 1 class mixed "IT and telecommunications" (INF3) and "transport and logistics" (TRAN) (students follow intersecting subjects together and split when following specific subjects); and 1 class of "chemistry, materials and biotechnology" (CHEM).

During the first two years (usually very similar, to allow pupils to change track), all students of a technological institute should have followed some introductory computing lessons (no more than 2-3 hours a week for one or two years), but the content varies a lot: from the use of office suites to visual programming in Scratch to web development. We asked students if they had already programmed, to check for potential effects.

During the third year, students in "IT and telecommunications" follow three courses related to Computer Science and Programming (Network systems, IT project management, Computer science) for a total of 13 hours a week. Students in "transport and logistics" and "chemistry, materials, and biotechnology" do not follow any CS-related course.

We asked for informed consent from students' parents, and only those who gave us back the approved informed consent participated in the study. As the study involved a pre and a post questionnaire, students were anonymously identified with a secret code (the name of a color) they had to remember. We report, in Table 12.1, number, gender, and previous programming experience of only students that completed both the pre and the post-questionnaire. Most of the students are males, and the females are almost all from the CHEM class. This is not surprising since technological tracks, especially those related to mechanics, electronics, and computer science, are usually chosen mainly by males in Italy.

12.2.2 Methods

We decided to measure some constructs (some belonging to general mindset theory, e.g., performance vs. mastery orientation - see Subsection 5.1.1, and some specifically related to CS mindset) at the beginning and at the end of the third year (school year 2017/2018).

Moreover, we decided to implement a small mindset intervention in the INF1 class, halfway through the year, to see if we could alter students' mindset.

12.2.2.1 Questionnaire

Research on mindsets is usually conducted by asking students to rate how much they agree (on a Likert scale, e.g., from 1 to 6) on some statements like "*You have a certain amount of intelligence, and you can't really do much to change it*" and then calculating a score representing students' mindset.

For traditional mindset constructs, robust and validated scales are available. We used the wording in Dweck [2003], an Italian translation of Dweck [1999], to measure:

- **Mindset**, obtaining a value from 1 (fixed mindset) to 6 (growth mindset);
- **Confidence in one's intelligence**, obtaining a value from 1 (low confidence) to 12 (high confidence);
- **Goal choice**, obtaining a value from 1 (performance orientation) to 6 (mastery orientation).

To measure belief constructs related to CS, we translated in Italian scales from Sun [2015], a Ph.D. thesis studying specifically Mathematical Mindsets, and adapted them by changing the word "Math" to "Computer Science"[3] in the original questionnaire. In particular, we measured:

- **Computer Science Mindset**, obtaining a value from 1 (fixed CS mindset) to 6 (growth CS mindset);
- **Beliefs about the nature of CS**, obtaining a value from 1 (CS as a fixed discipline, where you have to learn facts, rules, give quickly the *only* right solution to problems) to 6 (CS as a creative discipline, where you can learn from your errors and have many solutions to problems);
- **Identification with CS**, obtaining a value from 1 ("not a CS person", bad at CS) to 5 ("a CS person", good at CS);
- **Mastery orientation in CS**, obtaining a value from 1 (performance orientation in CS) to 5 (mastery orientation in CS).

In Appendix C.1, the full English translation of the questionnaire is reported, with details on how the scores are calculated.

The school was unable to authorize access to students' grades. However, we asked CS-oriented students if their grades in CS-related disciplines decreased, increased, or remained the same.

The questionnaire was administered to all classes in two days, one at the beginning and one at the end of the school year, in school laboratories, through Google Forms, by this author

[3] *Informatica*, in Italian

(who has no relationships with the school, and therefore presented himself as an entirely external subject, to avoid the tendency to socially desirable answers given by students to their teachers).

12.2.2.2 Intervention

In class INF1, halfway through the year, we performed a mindset intervention, attempting to alter students' beliefs by explicitly teaching them about growth mindset (analogously to what Dweck and colleagues did in many experiments: see, for example, Blackwell et al. [2007]). The intervention (a 2 hours lesson conducted by the author) was inspired by material prepared by Khan Academy and PERTS [2014]. The lesson included[4], in order:

- a discussion on what students think intelligence is, and if it could be changed;
- a discussion about a situation where they learned to do something they were not able to do yet;
- a presentation of the Mindset theory through a table translating of the diagram from Holmes [2016];
- a short video [Sentis, 2012] on neuroplasticity (with Italian subtitles);
- and a description by the researcher of "*a situation where he overcame a struggle in learning and learned to solve a problem*" [Khan Academy and PERTS, 2014].

In the last part of the lesson, three open questions were administered through a Google Form. Students were asked to *think of a time when they overcame a struggle to learn something* and then to:

OQ1 Advise future students on how to overcome an obstacle when learning something new [translation of question in Khan Academy and PERTS [2014]]

OQ2 Suggest what a teacher could do to stimulate a growth mindset in students [proposed by us]

OQ3 Suggest some concrete strategies to a future CS student to succeed and not be discouraged when facing difficulties in learning to program [inspired by Simon et al. [2008]]

The full wording of the questions is available in Appendix C.2.

[4] Slides used during the lesson are available at https://goo.gl/ZcBzX1.

Table 12.2: Internal consistency (Cronbach's alphas)

Measure		INF1	INF2	INF3	CHEM	TRAN	ALL
GM	pre	0.86	0.90	0.66	0.70	0.89	0.83
	post	0.94	0.97	0.93	0.82	0.26	0.89
CONF	pre	0.62	0.87	-0.34	0.81	0.49	0.67
	post	0.71	0.79	0.17	0.60	0.07	0.65
GOAL	pre	0.62	0.50	-0.21	0.46	-0.19	0.46
	post	0.35	0.29	-0.39	0.37	0.59	0.30
CSM	pre	0.60	0.80	0.57	0.71	0.52	0.72
	post	0.53	0.79	0.82	0.77	0.80	0.75
NAT	pre	-0.43	-0.21	0.13	-0.56	0.05	-0.08
	post	0.30	0.27	0.73	-0.52	0.51	-0.01
IDCS	pre	0.88	0.84	0.70	0.87	0.60	0.86
	post	0.91	0.77	0.76	0.83	0.74	0.85
MAST	pre	-0.43	0.69	0.81	0.35	0.49	0.40
	post	0.21	0.60	0.75	0.66	0.31	0.57

12.3 Results

12.3.1 Quantitative Analysis

Responses were analyzed through the statistical programming language R [R Core Team, 2013], with the packages *Hmisc*, *pastecs*, *lsr*, *psych*, *dplyr*.

Internal consistency was measured through Cronbach's alpha [Nunnally, 1978] for questions about mindset (GM), confidence in one's own intelligence (CONF), goal choice (GOAL), Computer Science mindset (CSM), beliefs about nature of CS (NAT), identification with CS (IDCS), mastery orientation in CS (MAST) for pre- and post-questionnaire responses, for each class and for the whole sample of students. Values are reported in Table 12.2.

For many measures the internal reliability is very low: it could be due to the relatively small sample size, or to the translation/adaptation to CS of some questions, or other factors. Note however that, in the inspiring study on mathematical mindset, even lower alpha values were found [Sun, 2015, e.g., pp. 220-224].

We decided to explore further with the analysis of GM, CSM and IDCS, which, according to Nunnally [1978], have acceptable (> 0.70) Cronbach's alphas for both pre and post-test for combined samples.

12.3.1.1 Difference Between Beginning and End of the Year

To check if studying or not studying CS had an impact, we performed statistical tests on the difference between pre and post scores for each class and each of the three measures. Moreover, we also performed the test for all the samples combined.

Paired samples t-test works if the differences between pre and post scores are normally

distributed. We checked that with the Shapiro-Wilk normality test and, since often it gives false-positive results for small samples, also checked normality graphically through QQ-plots. Whenever we were unsure about the normality of data, we performed a non-parametric version of the test, in particular, the Wilcoxon signed-rank test with continuity correction, that does not require a normality assumption. When we found significant results, we also measured the effect size through Cohen's d for paired samples. Results are reported in Table 12.3.

Mindset decreased in INF1, INF3, CHEM, and increased in INF2 and TRAN, but in no case was the change statistically significant. Regarding the combined samples, mindset decreased for both CS and non-CS students, but not significantly.

Identification with CS decreased in INF classes, and increased in CHEM and TRAN, but not significantly in any case. This reflects on the fact that it decreases for CS students combined, and increases for non-CS students (not significantly, but with $p = 0.10$).

Computer Science mindset decreased, but not significantly, in INF1 and INF2, remained stable in INF3, but decreased significantly ($p = 0.02$) in CHEM (with a medium effect size $d = 0.52$) and decreased ($p = 0.09$, with a large effect size $d = 0.87$) in TRAN. In the combined samples, CSM decreased, but not significantly, in CS students, and significantly in non CS students ($p < 0.01$, with a medium/large effect size $d = 0.59$).

12.3.1.2 Difference Between Different Subgroups

To check if there were significant differences between some subgroups, we performed many independent samples tests. In particular, we checked for differences in GM and CSM (at the beginning, at the end, and their difference) between males and females, between students with and without previous programming experience, and between students of CS and non-CS classes.

To perform an independent samples t-test, both subgroups must be normally distributed and have the same variance. All normal subgroups were checked with the F-test to compare the two variances and were found to have the same variance. Groups that were not normally distributed were analyzed with the Wilcoxon rank-sum test with continuity correction, which does not require normality assumption. Results are shown in Table 12.4.

We found no statistically significant differences in mindset between any of the subgroups. By contrast, we see significant differences in the difference between final and initial CS growth mindset. In particular, it decreased in females more than in males ($p = 0.16$), decreased much more in novices than in those with previous experience ($p = 0.06$) and, confirming previous observations, decreased statistically significantly more in non-CS students than in CS students ($p = 0.03$).

12.3.1.3 Correlation With Grades

To see if mindset influences CS grades, we calculated Pearson's r between GM/CSM (pre, post, diff) and the reported variation in grades, but did not find significant correlations, as shown in Table 12.5.

Table 12.3: Paired samples tests

Sample	Measure	n	normal	pre		post		t-test			Cohen's d	Wilcoxon-test	
				mean	sd	mean	sd	t	df	p		V	p
INF1	GM	16	yes	4.02	1.28	3.33	1.39	1.56	15	0.14			
	CSM	16	not sure	4.50	0.84	4.20	0.97					38.00	0.31
	IDCS	16	yes	3.28	0.96	3.07	0.94	1.02	15	0.32			
INF2	GM	14	yes	4.05	1.54	4.17	1.69	-0.36	13	0.72			
	CSM	14	not sure	4.68	1.02	4.38	1.12					58.50	0.38
	IDCS	14	not sure	3.71	0.65	3.68	0.83					47.00	0.94
INF3	GM	7	yes	4.29	1.15	4.05	1.46	0.35	6	0.74			
	CSM	7	not sure	3.50	0.80	3.50	1.14					8.50	0.89
	IDCS	7	not sure	3.83	0.46	3.75	0.55					10.00	0.59
CHEM	GM	23	yes	4.22	1.07	3.99	1.22	0.98	22	0.34			
	CSM	23	yes	4.21	1.11	3.71	1.12	2.51	22	0.02*	0.52		
	IDCS	23	yes	2.79	0.83	2.91	0.88	-0.83	22	0.42			
TRAN	GM	6	no	2.89	1.34	3.22	0.78					2.00	0.79
	CSM	6	yes	5.04	0.62	4.33	0.90	2.14	5	0.09	0.87		
	IDCS	6	not sure	2.59	0.45	3.20	0.44					1.00	0.06
ALL CS	GM	37	yes	4.08	1.33	3.78	1.53	1.14	36	0.26			
	CSM	37	no	4.38	0.98	4.14	1.08					259.50	0.20
	IDCS	37	yes	3.54	0.79	3.43	0.88	1.18	36	0.25			
ALL NON-CS	GM	29	yes	3.94	1.23	3.83	1.17	0.55	28	0.59			
	CSM	29	yes	4.38	1.07	3.84	1.10	3.18	28	0.00*	0.59		
	IDCS	29	yes	2.75	0.76	2.97	0.81	-1.71	28	0.10			

Table 12.4: Independent samples tests

	M		F		t/Wilcocxon			
n	56		10					
	m	sd	m	sd	t	W	p	d
GM pre	3.99	1.27	4.17	1.36	-0.39		0.70	
GM pos	3.70	1.40	4.40	1.12		194	0.12	
GM diff	-0.30	1.40	0.23	1.37	-1.11		0.27	
CSM pre	4.36	0.96	4.47	1.36		236	0.43	
CSM pos	4.06	1.04	3.70	1.33	0.96		0.34	
CSM diff	-0.30	0.96	-0.78	0.88		359	0.16	
	EXP		NO EXP		t/Wilcocxon			
n	30		36					
	m	sd	m	sd	t	W	p	d
GM pre	4.07	1.21	3.98	1.35		512	0.72	
GM pos	3.94	1.20	3.69	1.51	-0.76		0.45	
GM diff	-0.12	1.36	-0.30	1.44	-0.50		0.62	
CSM pre	4.28	0.99	4.46	1.04		593	0.50	
CSM pos	4.21	1.02	3.83	1.13	-1.41		0.16	
CSM diff	-0.08	0.91	-0.63	0.94		395	0.06	0.6
	CS		NO CS		t/Wilcocxon			
n	37		29					
	m	sd	m	sd	t	W	p	d
GM pre	4.08	1.33	3.94	1.23		496	0.60	
GM pos	3.78	1.53	3.83	1.17		544	0.93	
GM diff	-0.30	1.59	-0.12	1.12	0.52		0.60	
CSM pre	4.38	0.98	4.38	1.07	0.00		1.00	
CSM pos	4.14	1.08	3.84	1.10	-1.11		0.27	
CSM diff	-0.24	0.98	-0.54	0.92		371	0.03*	0.3

Table 12.5: Correlation between grades and mindsets

Grades					
GM	r	sig.	CSM	r	sig.
pre	0.04	0.80	pre	0.07	0.70
post	0.27	0.11	post	0.16	0.33
diff	0.22	0.19	diff	0.11	0.50

12.3.2 Qualitative Analysis

We analyzed the responses to open questions from the students in INF1. Although the questions asked them to write a letter or to give suggestions in at least five sentences, all responses were brief and schematic. This, however, made the coding procedure straightforward: we identified categories of advice and could easily fit answers in those categories.

We analyzed the responses (see § 12.2.2.2) of the 16 students of INF1 that also completed both the pre and the post questionnaires.

Categories and the number of suggestions fitting in that category are reported here. Note that many answers contained more than one piece of advice, so fit in more than one category.

OQ1 Categories - *advice to future students on overcoming obstacles*: put effort (4); do not discourage (2); practice a lot (2); be convinced before choosing CS (1); take private / extra lessons (2); ask teacher (1); decompose problems (1); ask online (1); (in the context of video-game programming) find and remember errors early to avoid them later (1).

OQ2 Categories - *suggestion for the teacher to stimulate a GM in students*: make students have fun (3); repeat things often (3); link to student interest/passion (3); limit boring explanation/theory (2); encourage group work (2); involve students in logic reasoning (2); deliver interactive lessons (1); avoid numeric grades (1); propose creative exercises (1); link subject to real life (1); teach value of effort and training (1); personalize teaching (1).

OQ3 Categories - *concrete strategies to overcome difficulties when learning to program*: program on your own at home constantly (9); watch online video tutorials/examples (6); constantly study (5); have objectives/interest/ motivation regarding CS (4); believe in yourself/ do not discourage (4); ask questions (4); pay attention during lessons (3); study the programming language (syntax) (3); understand fundamentals (2); take notes (2); not remain behind: concepts builds on previous ones (2); you can do everything with programming (1); study theory (1); there are many ways to solve a problem (1); reflect on errors (1); understand instead of learning by heart (1); break down the problem in sub-problems (1); read compiler error messages (1); try to engage yourself by programming video games (1).

12.4 Discussion, Conclusions, and Further Work

We found (RQ1) no statistical evidence that studying (or not studying) CS for one year automatically fosters a growth mindset in high school students. Moreover, we did not find significant differences between males and females, students with or without previous programming experiences, CS or non-CS students.

We see these results neither as surprising nor as negative, since they are in line with educational research, stating that transfer is difficult between distant domains and does not happen automatically (Chapter 4). Our results support warnings about enthusiastic claims around CS fostering a growth mindset.

We still think some characteristics of CS can help to develop a growth mindset, but only if teaching interventions are explicitly designed to do so (and if they leverage, for example, on the creative and iterative nature of CS, as we have successfully shown in a preliminary study, described in Chapter 13).

CSEd research should also *focus on the opposite implication*: CS is a challenging subject, therefore a growth mindset can be particularly important to succeed in it [Murphy and Thomas, 2008; Lewis, 2017].

The intervention where we explicitly taught about growth mindset in one of the CS classes (INF1) was not effective (RQ2): the average mindset of INF1 students decreased (although not significantly) from 4.02 out of 6 to 3.33 out of 6. One of our possible explanations is the fact that the initial discussion during the lesson really engaged students and took much more time than scheduled, forcing the researcher to rush through the following points. Furthermore, a shift towards a more fixed mindset was also found after a similar intervention in CS1 courses by Simon et al. [2008]. Finally, the efficacy of this kind of intervention is debated (see Subsect. 5.1.4).

However, INF1 students, after receiving the intervention, gave to hypothetical colleagues coherent GM suggestions (RQ3) to overcome difficulties (put effort, ask questions, practice regularly, and so on). Surprisingly, when asked what a teacher should do to stimulate a growth mindset, almost no one answered with typical GM teaching suggestions (e.g., teach explicitly about brain growth, praise effort): *the vast majority advocated more creativity, engagement, fun, connection with the real world or student passions*. This is interesting because, in other fields, the suggested approaches proved to be effective [Reid and Ferguson, 2014]. Our preliminary results on Primary Education students seem to agree with that (see Chapter 13).

We found (RQ1) a statistically significant decrease in "Computer Science mindset" only in students not studying CS. The decrease was higher for girls and students without any previous programming experience. This is not desirable: if we think CS has a universal social value, reinforcing stereotypes about a "geek gene" [Ahadi and Lister, 2013; Patitsas et al., 2016] will be harmful and lead to a problem similar to those many students are experiencing in Math, as international tests reveal. This adds to the evidence on the importance of introducing CS principles for all K-12 students.

Identification with CS decreased (RQ1) in CS students, confirming what was already observed for Math by Wigfield et al. [1991]. Its (small) increase in non-CS students is worth further research.

This research was conducted in a high school in Italy, with 16-year-old students. To generalize the results, it should be reproduced with different samples (different kinds of schools, different geographic areas, different ages and previous experience, different teaching methodologies, and so on). Items on CS beliefs were translated into Italian and adapted from recent Math research: thus, we need validation studies and better calibration to assess CS beliefs effectively.

Chapter 13

Creative Computing Could Foster a Growth Mindset[1]

As already discussed in previous parts of this dissertation, the introduction of CS in Pre-University education, as early as in pre-school and primary school, raises the need for extensive teacher training.

As discussed in Chapter 5, introductory programming courses are known to be difficult, and some studies suggest they foster an entity theory of intelligence (fixed mindset), reinforcing the idea that only some people have the "geek gene". This is particularly dangerous if thought by future primary school teachers, that can have a great impact on their students' views on CS.

In this Chapter, we report on an exploratory study in which we analyzed the effects of an introductory course about *Creative Computing with Scratch* for Primary education students, and observed a statistically significant increase of pre-service teachers' growth mindset while observing a statistically significant decrease in their computer anxiety.

The structure of the course is detailed, with particular emphasis on some characteristics that may have determined a growth mindset increase.

Limitations of this exploratory study are discussed, and future work is depicted.

13.1 Introduction and Motivations

Since 2014, at the University of Bologna, a laboratory course on *Creative Computing with Scratch* has been taught to pre-service primary teachers. During the first two years of teaching, instructors collected oral reports from students. At the beginning of the course, many of them reported anxiety and low self-efficacy about learning to program. Some of them described themselves as *not a computer science/technology person*. On the contrary, at the end of the course, instructors received very good feedback. Some students spontaneously thanked them because they did not feel any more like they were not *computer science people* and felt empowered to be creative with technology and to teach it to their future pupils.

[1] This Chapter is based on material published in Lodi [2018a]

Based on these anecdotal pieces of evidence, we decided to explore changes in mindset (Chapter 5) and computer anxiety (Section 13.2) before and after the fall term course of the academic year 2016-2017. The test on mindset was replicated in the spring term course.

13.2 Computer Anxiety

Computer anxiety can be defined as "*a fear of computers when using one, or fearing the possibility of using a computer*" [Sam et al., 2005], and differs from negative attitudes toward computers. In fact, it involves a more affective response: *resistance to and avoidance of computer technology are a function of fear and apprehension, intimidation, hostility, and worries that one will be embarrassed, look stupid, or even damage the equipment* [Heinssen et al., 1987].

Computer anxiety has been correlated with math anxiety and gender [Heinssen et al., 1987; Maurer, 1994]. Females were found to have higher computer anxiety. By contrast, previous exposure to computers is correlated with a low level of anxiety. As Maurer [1994] suggests, females could have less exposure to computers than males due to stereotypes, so previous exposure should be taken into account.

A study by Martocchio [1994] correlates computer anxiety and implicit theories. It showed that computer anxiety decreased in participants of a basic computer training course who were taught incremental conceptions of ability, while did not change in participants to whom fixed entity conceptions of ability were taught.

To assess computer anxiety, the *Computer Anxiety Rating Scale (CARS)* was developed and validated by Heinssen et al. [1987].

13.3 The Study

We decided to measure growth mindset and computer anxiety changes between the beginning and the end of the laboratory course.

We decided to not teach explicitly about growth mindset or brain growth, performing what Yeager and Walton [2011] call a "stealthy intervention": the instructor (the author of this dissertation) paid particular attention in avoiding explicit mentioning of Dweck's research and ideas in lessons/suggested reading material, to avoid influencing the subjects and to test if creative computing and learning could influence growth mindset.

No active intervention was made to influence computer anxiety.

13.3.1 The Context

Currently, to become a pre-school and/or a primary school teacher in Italy, you have to get a 5-year (Single cycle/Combined Bachelor and Master) Degree in *Primary teacher education*. When graduating, students also get a *Pre-school and Primary school teaching license* that allows them to teach in Italian public schools.

For historical and sociological reasons, in Italy, primary teachers are mainly females, and this is reflected in the fact that *Primary teacher education* students are almost all females as well (for instance, 91% in A.Y. 2016/17 in our University).

At the University of Bologna, Primary Teacher education students take a mandatory *General Education and Educational Technologies* exam during the first year, and follow a hands-on 24 hours (3 credits) *Educational Technology Laboratory* during the fourth year. For that course, they can choose between several topics, from the use of interactive white-board, to stop-motion storytelling techniques and many others, all with the aim to learn how to use technology as a tool for effective teaching. To allow students to be supported by instructors and to work with technologies actively, each thematic-laboratory has a maximum of 32 students. In the context of this *multi-track course,* since the academic year 2014-15, students can choose the *Laboratory of creative computing*, to learn the basics of programming and creative computing with Scratch. To give the opportunity to take the course to more students (of the same academic year), the laboratory is replicated (same instructor, schedule, location, contents) in the fall term and in the spring term.

13.3.2 The Course

The course was designed as an introduction to creative computing with Scratch (Subsec. 8.3.6.1), following creative learning principles (Sec. 3.3). It was made up of 6 lessons, 4-hours each. The course plan is now described. Many activities are taken from MIT/Harvard creative computing materials (see Subsec. 8.3.6.1).

Lesson 1

- Brief introduction to creative computing and computational thinking;
- experiments with Google Presentations: the teacher creates a shared presentation with writing rights, then she asks students to add a new page and to write something about themselves, putting on a photo, and so on. This activity is initially messy, but soon students learn in a very bottom-up fashion to use the tool and to avoid modifying peers content;
- free exploration of Scratch;
- mini challenge to make something happen with it (*Scratch surprise* [Brennan et al., 2014]);
- guided tutorial to create a simple video game that contains a lot of computational concepts[2].

[2] A simplified version of Carmelo Presicce's *Under the sea*: https://scratch.mit.edu/projects/14759947/

Lesson 2

- Ten blocks challenge[3];
- free artistic project with just the simple hint to use the *pen* and *looks* categories blocks;
- debug exercises: students had to choose some debug exercises from Brennan et al. [2014], remix them, find the bug and comment out how they found it and what they did to correct it.

Lesson 3

- Witness from an invited primary school teacher using unplugged activities and Scratch in her teaching;
- examples of Scratch projects to be used with pupils[4];
- *about me* [Brennan et al., 2014] free project: create a project to introduce yourself and the things you do and love.

Lesson 4

- Exploration of *ProgrammaIlFuturo* (Subsec. 1.4.5) / Code.org (Subsec. 1.3.1) and comparison with Scratch: try some activities, find out pros and cons of the different platforms and approaches;
- advanced features (cloning and webcam): try to reproduce a *snowing-like* behavior with cloning and catching the clones with the hand through webcam-sensing[5].

Lesson 5

- Scratch and the physical world (Makey Makey[6] and Lego WeDo[7]) demos;
- time to work in small groups on the final project.

Lesson 6

- Public presentation of final projects: design of a teaching activity with Scratch for Primary School in the light of creative learning's 4 Ps (described in Section 3.3). Students were advised to be particularly careful not to create a game or a project on which their pupils would have been passive consumers, but rather design a creative

[3] https://creative-computing.appspot.com/unit?unit=4&lesson=13
[4] e.g. from https://scratch.mit.edu/studios/1918506/
[5] e.g. https://scratch.mit.edu/projects/129283065/
[6] e.g. the classical *human chain* and *whack a mole*: https://scratch.mit.edu/projects/43681296/
[7] e.g. simple sensor/motor use with Scratch described in: https://www.youtube.com/watch?v=qBhIcb-Ipmw

activity where their pupils should be free to create different projects but in the context of some Primary school teaching objectives of one or more subjects.

Each student had her own PC, but they were allowed to work in pairs/small groups, to get up and move around the laboratory and to communicate with each other. The teacher was always available to offer help.

The laboratory was mainly hands-on: students were assigned projects with a broad theme (e.g., *a project about you*) and given time to freely create with Scratch, experimenting, getting help online or asking the instructor. Sometimes, more structured exercises were given, but they were chosen not to be mechanical or repetitive. Instead, they were explicitly posed as challenges to have fun with, while learning. No theoretical lectures about programming or Scratch were given. However, some tips and quick demos, always *after* they worked a while on projects or problems, were given.

Homeworks consisted of realizing other projects at home and writing a page in a shared online notebook (Google Presentation), reflecting on difficulties, achievements, and learning process. The instructor gave feedback as comments, and students were invited to comment at least two of their mate's pages each week.

An online virtual class was set up with a Google+ community, where students could ask for help (to mates and to the teacher) and discuss. The instructor posted interesting videos/articles and stimulated comments.

The exam was pass/fail, with no grades. At the beginning of the course, it was clearly stated that students would have been evaluated for their effort (measured with the presence in class, participation, shared projects) rather than on the quality of their works. The final presentation of a group project was also mandatory to pass the exam. Students were particularly encouraged to share their projects even if they were buggy or incomplete, and ask for help.

All students passed the exam in both terms. Projects and journals were not graded but were checked by the instructor, and written feedback was given.

No explicit reference to growth mindset and brain growth theories were made. However, other growth mindset strategies (see Subsec. 5.1.2) were put in action: in particular it was carefully paid attention to give *growth mindset feedback* (both oral and written in the comments to projects or posts) and it was *praised process and outcome* (*You worked a lot to create that!*, *Very good project!*) rather than the person (*Bravo!*, *You are very good at it!*).

The instructor established a good class climate, where errors were not stigmatized but seen as a powerful tool for learning, helping students reflect on them with guided questions rather than simply *tell the solution*. This was eased by the tool: as noted in Section 8.3.6.1, Scratch helps you not be frustrated by errors and instead motivates you to figure out how to fix your bugs. Scratch tinkerability also helped to encourage students not to give up, moving forward by trial and error, and feeling empowered by their learning and successes.

13.3.3 Data Collection

An identical survey was administered at the beginning (pre-survey) and at the end of the course (post-survey). It was divided into two sections (Growth Mindset and Anxiety) in the

fall term course, while had only the Growth Mindset section in spring term one.

The first section was intended to assess students' mindset through the Implicit Theories of Intelligence Scale (see Appendix D.1.1 for the full scale). Similarly to the shorter version used in the study reported in Chapter 12, it included eight Likert-type items [Dweck, 1999, p. 285]. Students were asked to rate from 1 (*completely disagree*) to 6 (*completely agree*) eight statements about intelligence (in Italian in the survey), four reflecting an incremental theory, like *No matter who you are, you can significantly change your intelligence level* and four reflecting an entity theory, like *You have a certain amount of intelligence, and you can't really do much to change it*. During the analysis, the latter were reverse-scored, so that high points were associated with a growth mindset, while low points with a fixed mindset.

The second section was intended to assess student's anxiety through the Computer Anxiety Rating Scale (see Appendix D.1.2 for the full scale). It included nineteen Likert-type items, proposed by Heinssen et al. [1987] (we translated in Italian the slight variation of the original statements proposed by Sam et al. [2005]). Students were asked to rate from 1 (*completely disagree*) to 5 (*completely agree*) proposed statements (in Italian in the survey), half of them reflecting a high anxiety (like *I have avoided computers because they are unfamiliar and somewhat intimidating to me* or *I do not think I would be able to learn a computer programming language*) while half of them reflecting a low level of anxiety (like *Learning to operate computers is like learning any new skill, the more you practice, the better you become*). During the analysis, the latter were reverse-scored, so that high points were associated with high anxiety, while low points with low anxiety.

The pre-survey was administered right at the beginning of the fall term course, when students knew only the title and a very brief description of the course from the website. The post-survey was administered at the end of the course. Five weeks passed between the two administrations.

The questionnaires were anonymous, but answers of the same subject to pre and post-questionnaire were linked with a randomly generated code unknown to the researcher. The questionnaires were administered with a Google Form.

This process was repeated for the spring term course, but with mindset questions only.

A total of twenty-three students ($N = 23$), all females, aged from 21 to 29 ($M = 23$, $SD = 1.95$, $MODE = 22$) completed both the pre-survey and the post-survey of the fall term course.

Moreover, a total of twenty students ($N = 20$), all females, aged from 22 to 28 ($M = 23.05$, $SD = 1.82$, $MODE = 22$) completed both the pre-survey and the post-survey of the spring term course.

13.4 Data Analysis and Results

The data were analyzed with the R programming language [R Core Team, 2013].

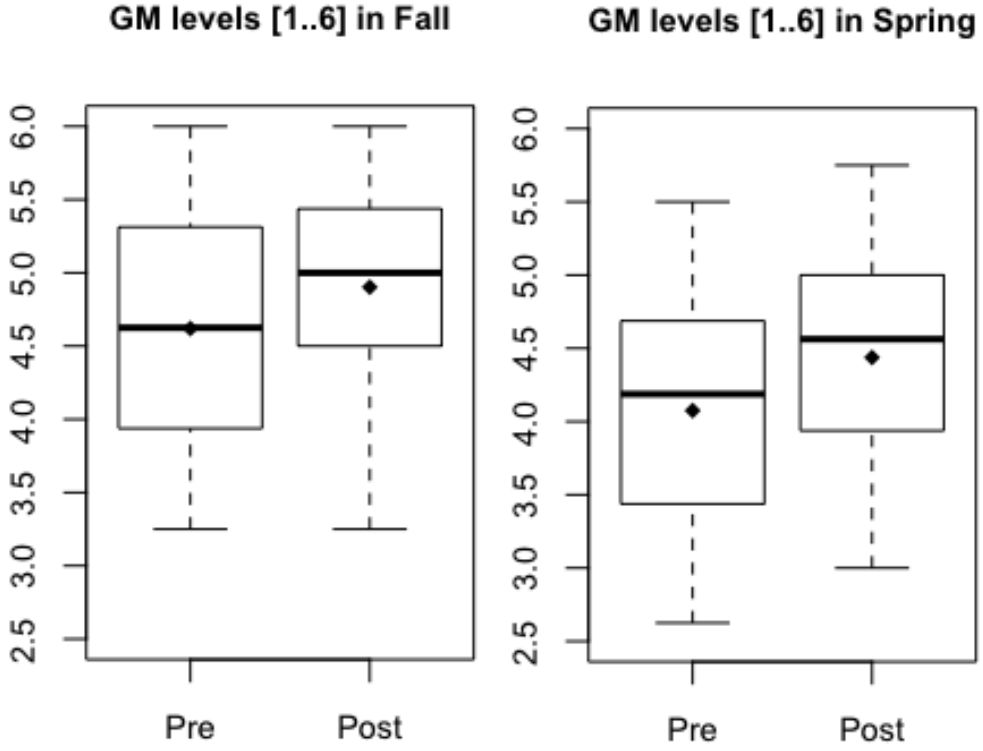


Figure 13.1: Distribution of growth mindset levels before and after fall and spring term course

13.4.1 Growth Mindset

For each subject, the initial and final growth mindset level was calculated. The growth mindset level is a value from 1 (fixed mindset) to 6 (growth mindset), calculated as the mean of the eight answers (with entity items reverse-scored, as stated) of each subject.

A paired-samples t-test was conducted to compare growth mindset at the beginning and at the end of the fall term course. There was a statistically significant difference in the mindset scores between the pre-test ($M = 4.62$, $SD = 0.78$) and the post-test ($M = 4.90$, $SD = 0.76$): $t(22) = -2.35$, $p = 0.028$ (< 0.05). In particular, growth mindset has increased from the beginning to the end of the course. (see Fig. 13.1, where mean GM of all subjects is represented as a black diamond).

Moreover, a paired-samples t-test was conducted to compare the growth mindset at the beginning and at the end of the spring term course. Again, we found a statistically significant increase in the mindset scores between the pre-test ($M = 4.08$, $SD = 0.80$) and the post-test ($M = 4.44$, $SD = 0.83$): $t(19) = -2.50$, $p = 0.022$ (< 0.05) (see Fig. 13.1).

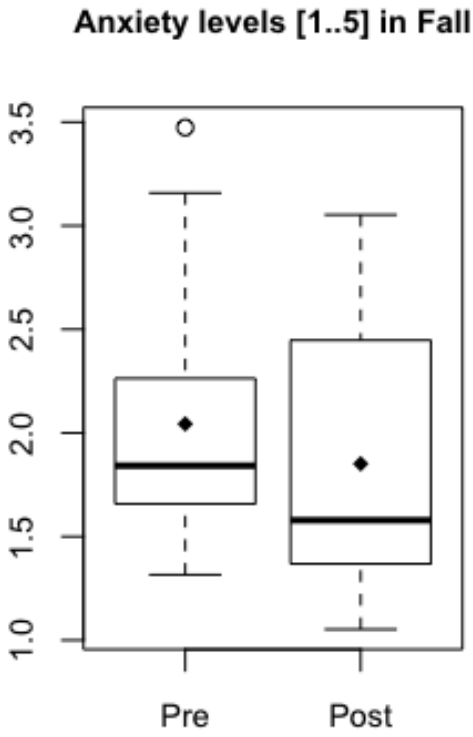


Figure 13.2: Distribution of anxiety levels before and after fall term course

13.4.2 Computer Anxiety

For each subject, the initial and final computer anxiety level was calculated. Anxiety level is a value from 1 (low anxiety) to 5 (high anxiety), calculated as the mean of the nineteen answers (with some items reverse-scored, as stated) of each subject.

A paired-samples t-test was conducted to compare computer anxiety at the beginning and at the end of the course. There was a statistically significant difference in the anxiety levels at the beginning ($M = 2.04$, $SD = 0.58$) and at the end ($M = 1.85$, $SD = 0.60$): $t(22) = 2.98$, $p = 0.007$ (< 0.01). In particular, computer anxiety has decreased from the beginning to the end of the course (see Fig. 13.2).

13.4.3 Data Validity

All answers showed acceptable or high internal consistency [Nunnally, 1978], as reported in Table 13.1.

To be valid for a paired t-test, the distribution of the differences between the two related values of each subject should be approximately normally distributed. Differences from both growth mindset and anxiety scores passed the Shapiro-Wilk normality test, or the Wilcoxon signed rank test with continuity correction.

Table 13.1: Internal consistency of answers

Dataset	Cronbach's alpha
Growth Mindset (pre-surv. fall)	0.86
Growth Mindset (post-surv. fall)	0.86
Growth Mindset (pre-surv. spring)	0.76
Growth Mindset (post-surv. spring)	0.90
Anxiety (pre-survey)	0.90
Anxiety (post-survey)	0.92

Finally, both measures are resistant to test-retest [Heinssen et al., 1987; Dweck et al., 1995].

13.4.4 Limitations of the Study

Since it is a pre-experimental design, this study is afflicted by some limitations. In particular, there is no control group, and so we don't know if the course is the *only* or the *main* cause of the difference between results, or if external factors may have intervened; as a positive observation, anyway, both fall and spring groups registered a statistically significative increase in mindset.

Moreover, the sample was relatively small and not randomized: it was made up of students that decided to attend the class.

Finally, responses on the post-questionnaire may have been influenced by the experience of taking the test itself.

13.5 Discussion, Conclusions, and Further Work

Despite the limitations of the study, we found a statistically significant increase in participants' growth mindset (result replicated in the following identical course) and a statistically significant decrease in participants' computer anxiety.

The initial growth mindset was already high. From one side, this is not surprising, as, in their education studies, students most probably (and hopefully) have been exposed to the idea that intelligence is not a fixed trait. This is confirmed by the fact that Education is a field whose only a small percentage of practitioners believe that innate talent is the main requirement for success (while CS has a much higher percentage, for example) [Leslie et al., 2015]. However, this clearly contrasts with oral reports about their low confidence collected at the beginning of the course. As seen, researchers think one can have different mindsets in different areas: probably, these students have a high growth mindset in general, but hold fixed ideas about CS in particular. Very recent studies also questioned the adherence between questionnaire scores and students talking or behaviour [Gorson and O'Rourke, 2019]. Another problem is the high "social desirability" of the answers: even though questionnaires were

anonymous, it is not easy to admit (especially in the context of a University course) that you think you are not able to learn something.

More surprising is students' medium/low initial anxiety. This may be due to their young age: they grew up in a world where technologies are everywhere, so they rapidly get used to them. It may be the case to construct an updated anxiety scale, which considers this diffusion and tests anxiety about more profound skills related to CS/CT rather than computers themselves. It would also be interesting to measure CS self-efficacy.

Many mindset interventions explicitly teach about brain growth to influence self-theories about intelligence. Even though we recognize this can be important, we aimed to foster a growth mindset mainly with teaching innovations and fundamental aspects of computer science.

The aim of our exploratory study was mainly to test whether our insights about CS, creative computing, and growth mindset were correct. Now we have to:

- design a proper experiment to confirm these preliminary data;
- investigate more deeply what factors of our approach (both the CS-specific ones and those more connected to creative learning and computing) were the most significant to foster a growth mindset: we believe that instructor's feedback, focus on iterative approach, debug, trial end error, open projects, and engaging exercises were crucial;
- investigate specific *computer science mindset*, both with ad-hoc scales and with other inquiry methods like interviews and direct observations;
- reconcile the focus on creative learning with the importance of teaching domain-specific CS core concepts, which is one of the important points this whole thesis makes.

Part V

Conclusions, Appendix and Bibliography

Chapter 14

Conclusions

14.1 Results

In this thesis, we considered historical, epistemological, pedagogical, cognitive, and affective aspects that can have positive or negative effects on the introduction of CS in K-12 education.

As briefly anticipated in the Introduction, we found support for important claims regarding these aspects, bringing contributions to CSEd research.

CS should be introduced in K-12 education as a tool to understand and act in our digital world, and to use the power of computation as a tool for thinking and meaningful learning. CT is the (important) conceptual sediment of that learning. We designed a curriculum proposal in this direction. We analyzed some of the most important definitions of "computational thinking" proposed in the literature, finding that they share a lot of common elements of very different natures (Chapter 2). Some are mental processes (e.g., problem solving, problem decomposition, abstraction, logical thinking) or transversal competences (e.g., tolerance for ambiguity, perseverance) that resonate with the current narrative on the importance of the 21st-century skills, and are probably even one of the reasons of the widespread of CT in education. This is confirmed by large scale qualitative study we conducted (Chapter 10), showing that teachers mainly find value in introducing CS/CT in schools for reasons like promotion of awareness and comprehension of problem solving, logical thinking, creativity, attention, planning ability, motivation for learning, students interest, cooperation.

However, reviewed educational research warns about the transferability of this kind of higher-order thinking skills between disciplines, and some even doubt their teachability (Chapter 4).

Moreover, putting too much focus on this aspects risks to dilute the fundamental concepts that distinguish CS from other disciplines (e.g., the presence of a precise external executor that solves problems following provided algorithms, the possibility to describe and execute abstractions through specific languages, the possibility to simulate worlds, and so on). The definitions of CT contain many elements directly linked to CS methods (e.g., automation, data analysis, evaluation) and programming practices (e.g., iterating, debugging). However,

the vast majority of teachers we polled fail to include references to these CS specific aspects in their definition of CT (Chapter 11).

At the same time, a historical analysis (Chapter 6) of the original context in which the expression CT was firstly used (the constructionist approaches for teaching sciences with LOGO programming language) shows a similar path. Simplistic claims (rebutted by experimentations) on learning to program automatically improving (general) thinking skills overshadowed the powerful ideas behind the approach: using programming as a meta-tool for learning by constructing meaningful artifacts and directly experimenting in the conceptual but interactive worlds we can create with computers. Based on this historical and philosophical research, we argue that the introduction of CS in K-12 should be motivated more by the need of understanding and being active participants in the digital world we live in, both through the acquisition of a CS "disciplinary way of thinking" and through the use of the unique interdisciplinary power of CS expressive tools, rather than advocating not proved claims about transfer of these competencies to domains very far from CS.

Moreover, these argumentation forms the basis for an Italian curriculum proposal (Chapter 7), encompassing CS core concepts like algorithms and programming, together with objectives to teach CS as a powerful personal expressive medium.

The use of expressions like "computational thinking" (which is useful to make clear that we are not talking about digital literacy) and "coding" can cause misconceptions and partial understandings between non-specialist teachers, often focusing on unverified claims about transfer to general thinking skills, hence in need of specific training, both on pedagogical aspects and, first of all, on CS disciplinary content. Teachers' responses in our questionnaires (Chapter 11) seems to confirm that use of term "computational thinking" is useful in clarifying that we are not advocating for teaching how to *use* technology (while the term "Informatics," at least in the Italian school system, was - and still is - often associated with training in ICT literacy).

As already mentioned, however, non-specialist teachers tend to retain only the more domain-general aspects and forget the fundamental CS-specific core concepts.

The same can be said for the buzzword "coding" to indicate CT activities, that creates misconceptions (e.g., teachers considering it "more abstract and general than programming" or, conversely, "toy programming") and risk to turn CT away from its CS specific nature.

This is why it is fundamental (and advocated by teachers themselves) to provide specific training initiatives that focus not only on pedagogical knowledge, but also on CS disciplinary content knowledge, necessary to fully appreciate *CS big ideas*.

Programming practices, teaching tools like visual programming environments, and unplugged activities have some intrinsic constructivist and constructionist characteristics (e.g., visualization, sharing and remixing, creative construction of interactives, concept reconstructions, personalization, use of the body) that can be used to facilitate CS learning, as shown with activities examples. Constructivist teaching, despite some harsh criticisms, is one of the most advocated paradigms in today's education (Chapter 3). Educational research seems to point out that, if carefully designed and delivered (by

providing optimal guidance and scaffolding, and finding the right time to "tell" students what they need to know - e.g., when they feel the need after personal exploration and struggle), constructivist approaches have positive effects on students retention, motivation, and transfer.

We argue (Chapter 8) that programming has an intrinsically constructionist nature (it involves the creation of a sharable artifact). Moreover, modern visual environments like Scratch provide cognitive-constructivist characteristics (e.g., personification, visualization of the underline "notional machine"), social-constructivist characteristics (focus on sharing, remixing, collaborating on projects) and constructionist characteristics (engagement and motivation to creatively build meaningful artifacts). Analogously, unplugged activities (those not using a computing device) provide constructivist aspects (e.g., re-discovery of algorithms and solutions) and - despite clearly differing - also constructionist ones (e.g., kinesthetic approach to teaching complex abstract ideas). We note, however, how transfer from unplugged activities can happen only when concepts are explicitly linked with actual implementations through "plugged" activities. Moreover, iterative methodologies like agile software development share elements with the incremental approaches for creatively learning computing.

In light of these observations, we produced lesson plans using a "unplugged+plugged" approach (Chapter 9). We designed unplugged activities that induce students to construct fundamental CS and programming concepts personally, and then linked what they learned to plugged activities with the visual programming language Scratch, providing both open-ended creative experiences and more scaffolded activities to learn Math concepts. The activities are examples of how to teach for the objectives we indicated in our curriculum proposal.

Like other general skills, growth mindset is not automatically fostered by learning CS. Worse, not studying CS can foster fixed beliefs about CS ability. A growth mindset could, however, be fostered by creative computing activities, leveraging on the constructivist aspects of CS. We found (Chapter 12) that malleable beliefs about one's abilities (growth mindset) are not automatically fostered by studying CS in high school. Moreover, not studying CS can foster fixed beliefs about CS ability, which is not desirable since we value CS as a subject that everyone should learn.

However, a creative learning approach to teach computing can be effective in increasing malleable views of intelligence in pre-service primary teachers (Chapter 13).

Finally, more active, creative, engaging, constructivist and constructionist approaches (sharing elements with CS, as seen) are advocated by high-school students as methodologies that teachers should follow in order to foster their growth mindset.

14.2 Limitations

As a general limitation, this work can be seen as a bit unbalanced on the "debunking" side, showing that many common-sense ideas about computational thinking and transfer are not backed by science. This is mainly an effect of the intense outreach and training activities the author carried out before and during his Ph.D., that exposed him to a lot of that 'gray literature' like school blogs and educators online groups. That is how most of the teachers - not used to read scientific articles - find their source of information, and this subsequently

informs their teaching. Much worse: also politics (at least in Italy) often falls in these traps. It is, therefore, the duty of academics, possibly holding experience both in the CS disciplinary content and in pedagogical issues, to pave the way for a successful introduction of CS in K-12.

Active, creative, and constructive approaches, sharing elements with CS and probably having good effects on learning, transfer, and mindset, seem good candidates to explore further. This should be tested with small, focused action research, and then on a larger scale, also as a way to validate our proposed curriculum.

Studies on teachers' sentiment and misconceptions have a large sample, which, however, was not randomly selected: it is made up of people that, for any reason, are taking part in the Programma il Futuro project. Despite the fact that these subjects should be more conscious and aware of CT related topics, our results on their conceptions about CT and coding are not so encouraging. We suspect that teachers not interested in the project (and, we could infer, not interested in teaching CS in schools) could perform even worse.

The study on mindset between high school students did not consider teaching methodologies, that, research tells us, can have an impact on mindset.

Finally, the study on pre-service teachers is a pre-experimental design. Despite giving (repeated!) positive results, it should be re-implemented with a focus on the specific CS mindset. Moreover, since it was used a very open teaching methodology (creative learning), we need to assess what CT / CS core concepts students learned. This is not easy, because there is no agreement on what and how to assess with respect to CT learning objectives.

14.3 Future Work

There are a lot of possible paths to follow from these results.

It is fundamental to find proper ways to train teachers in both Content Knowledge and Pedagogical Content Knowledge about CS, to address their misconceptions. Finding and addressing misconceptions about terms like "computational thinking" and "coding" is important, to be sure the effort of introducing CT in K-12 education is made in the direction supported by the scientific community. However, it is also essential to focus on student learning outcomes concretely.

This is why another crucial point is to design activities that can address CS learning objectives, while maintaining an active, open, creative, constructivist, and constructionist approach, keeping together Wing's CT and Papert's CT. These activities must be objectively evaluated, and this poses many problems, for example, finding evaluation instruments and research methodologies suitable for such open-ended activities.

Learning tools are another area of research: for example, visual versus textual languages, and the transition between the two (possibly also mediated by new paradigms like "frame-based" programming).

Moreover, it is crucial to conduct studies on the relationship between modern environments and transfer of learning (from CS to near or far domains), both to verify (or falsify) the still very open hypothesis, and to design and test explicit ways of fostering transfer.

For what concerns mindset studies, it is necessary to find suitable instruments (validated, soundly translated) and tools to measure "CS growth mindset" accurately. These instruments should not be based only on self-reported levels of mindset (since these instruments have been recently questioned) but also on observable behaviors during programming activities (e.g., deleting all the code instead of trying to find the bug). This could then be useful to give growth mindset oriented feedback to users during their programming activity, and also to give them context-dependent hints about strategies they can adopt (the problematic part in fostering a growth mindset).

Appendix A

Other CT Definitions and Classifications

Partially following the review from Juškevičienė and Dagienė [2018, pp. 269-271], we report definitions or categorizations exisiting main components/skill/characteristics of CT found in recent literature.

Grover and Pea [2013] reviewed CT literature, assumed the Aho-Cuny-Snyder-Wing definition and agreed that the following elements are accepted in literature:

- Abstractions and pattern generalizations (including models and simulations)
- Systematic processing of information
- Symbol systems and representations
- Algorithmic notions of flow of control
- Structured problem decomposition (modularizing)
- Iterative, recursive, and parallel thinking
- Conditional logic
- Efficiency and performance constraints
- Debugging and systematic error detection

Selby and Woollard [2013], in a widely referenced Technical Report, examined a number of CT definitions, and argued that the most relevant and useful elements are:

- thought process,
- abstraction,

- decomposition,
- algorithmic thinking,
- evaluation,
- generalization.

Juškevičienė and Dagienė [2018], after reviewing many definitions, from Papert to those proposed up to 2017, found eight CT component groups.

- Data analysis & representation
  - Data collection
  - Data analysis
  - Data representation
  - Generalisation
  - Patterns finding
  - Drawing conclusions
- Computing Artefacts
  - Artefact development
  - Artefact designing
- Decomposition
  - Breaking into parts
- Abstraction
  - Details suppression
  - Modelling
  - Information filtering
- Algorithms
  - Sequence of steps
  - Procedures
  - Set of instructions
  - Automation
- Communication & collaboration
  - Communication

  - Collaboration
  - Computational analysis
- Computing & Society
  - Computing influence
  - Computing implication
  - Computing concepts
- Evaluation
  - Evaluation
  - Correction

Weintrop et al. [2016] proposes a "*definition of computational thinking for mathematics and science in the form of a taxonomy consisting of four main categories: data practices, modeling and simulation practices, computational problem solving practices, and systems thinking practices.*"

From a literature they start with an initial set of activities:

1. Ability to deal with open-ended problems
2. Persistence in working through challenging problems
3. Confidence in dealing with complexity
4. Representing ideas in computationally meaningful ways
5. Breaking down large problems into smaller problems
6. Creating abstractions for aspects of problem at hand
7. Reframing problem into a recognizable problem
8. Assessing strengths/weaknesses of a representation of data/representational system
9. Generating algorithmic solutions
10. Recognizing and addressing ambiguity in algorithms

After that, by analyzing CT activities for math and science, propose the "Computational thinking in mathematics and science taxonomy".

- Data practices:
  - Collecting Data
  - Creating Data

  - Manipulating Data
  - Analyzing Data
  - Visualiging Data
- Modeling and simulation practices:
  - Using Computational Models to Understand a Concept
  - Using Computational Models to Find and Test solutions
  - Assessing Computational Models
  - Designing Computational Models
  - Constructing Computational Models
- Computational problem solving practices:
  - Preparing Problems for Computational Solutions
  - Programming
  - Choosing Effective Computational Tools
  - Assessing Different Approaches/Solutions to a Problem
  - Developing Modular Computational Solutions
  - Creating Computational Abstractions
  - Troubleshooting and Debugging
- Systems thinking practices:
  - Investigating a Complex System as a Whole
  - Understanding the Relationships within a System
  - Thinking in levels
  - Communicating Informations about a System
  - Defining Systems and Managing Complexity

Shute et al. [2017] provide, after an extensive literature review, a very general definition of CT: "*the conceptual foundation required to solve problems effectively and efficiently (i.e., algorithmically, with or without the assistance of computers) with solutions that are reusable in different contexts.*". They then recognize the following categories and subcategories, giving explainations that are quite general far from CS.

- Decomposition
- Abstraction
  - Data collection and analysis
  - Pattern recognition

    - Modeling
- Algorithms
    - Algorithm design
    - Parallelism
    - Efficiency
    - Automation
- Debugging
- Iteration
- Generalization

Kalelioğlu et al. [2016] view CT as a "*complex higher-order thinking, skills may require to use the power of human cognitive ability and embrace the support of machines to think and solve problems.*" They propose a "Framework for Computational Thinking as a Problem-Solving Process".

1. Identify the problem
    - Abstraction
    - Decomposition
2. Gathering, representing and analysing data
    - Data collection
    - Data analysis
    - Pattern recognition
    - Conceptualising
    - Data representation
3. Generate, select and plan solutions
    - Mathematical reasoning
    - Building algorithms and procedures
    - Parallelisation
4. Implement solutions
    - Automation
    - Modelling and simulations
5. Assessing solutions and continue for improvement

- Testing
- Debugging
- Generalisation

Krauss and Prottsman [2016] define[1] CT as using thinking patterns and processes to pose and solve problems or prepare programs for computation. Lessons plans are given for the following categories:

- Decomposition (data analysis)
- Pattern matching (data visualization)
- Abstraction (data modelling, pattern generalization)
- Automation (algorithm design, parallelization, simu- lation)

College Board [2017] proposes a CT framework for AP CS Principles course, propsoing six practices:

1. Connecting Computing
    - Identify impacts of computing.
    - Describe connections between people and computing.
    - Explain connections between computing concepts.
2. Creating Computational Artifacts
    - Create a computational artifact with a practical, personal, or societal intent.
    - Select appropriate techniques to develop a computational artifact.
    - Use appropriate algorithmic and information management principles.
3. Abstracting
    - Explain how data, information, or knowledge is represented for computational use.
    - Explain how abstractions are used in computation or modeling.
    - Identify abstractions.
    - Describe modeling in a computational context.
4. Analyzing Problems and Artifacts
    - Evaluate a proposed solution to a problem.
    - Locate and correct errors.

[1] As cited in Juškevičienė and Dagienė [2018, pp. 270]

- Explain how an artifact functions.
- Justify appropriateness and correctness of a solution, model, or artifact.

5. Communicating

   - Explain the meaning of a result in context.
   - Describe computation with accurate and precise language, notations, or visualizations.
   - Summarize the purpose of a computational artifact.

6. Collaborating

   - Collaborate with another student in solving a computational problem.
   - Collaborate with another student in producing an artifact.
   - Share the workload by providing individual contributions to an overall collaborative effort.
   - Foster a constructive, collaborative climate by resolving conflicts and facilitating the contributions of a partner or team member.
   - Exchange knowledge and feedback with a partner or team member.
   - Review and revise their work as needed to create a high-quality artifact.

Denning and Tedre [2019] proposed the following definition:

> Computational thinking is the mental skills and practices for
> - *designing* computations that get computers to do jobs for us, and
> - *explaining* and interpreting the world as a complex of information processes.

Appendix B

Proposal for a national Informatics curriculum in the Italian school

This appendix is the full English version of the report by Nardelli, Forlizzi, Lodi, Lonati, Mirolo, Monga, Montresor, and Morpurgo [2017].

It is reported here mainly as a way to quickly reference to objectives.

B.1 Foreword

This proposal of the Italian academic Informatics[1]community aims at contributing to the development of Informatics education in the primary and secondary school in Italy.

The role of Informatics in school curricula is currently a topical issue of the education policy all over the world. In the US, the 2015 the "Computer Science for All" initiative puts Computer Science in schools on a par with other scientific and technological fields. In the UK, starting from s.y. 2014-15 Computing is a mandatory subject for all instruction levels. Similar démarches are under way in several other countries.

This document is the outcome of a long process, promoted by the Italian Informatics community, but that has also benefited from important contributions of pedagogists and experienced school teachers who took part in the discussion. For convenience, the proposal was drafted according to the model of related Italian Ministry of Education, University and Research (MIUR) documents reporting curricular indications. In addition, it attempts to explain as clearly as possible our cultural and scientific standpoint.

On the one hand, according to our community's view, Informatics is an independent scientific discipline that provides the concepts and the languages necessary to understand and to fully participate in the digital society. On the other one, it is a cross-disciplinary field that offers an additional point of view to interpret phenomena and to approach problems.

We are nevertheless aware that the current outcome is just the first step of a long journey and that our task cannot yet be considered as accomplished. Further opportunities for

[1]In the translation of the proposal, the term "Informatics" was preferred, since it is closer to the italian "Informatica". See the terminological note in the Introduction.

discussion with teachers, experts in education and people in charge of school policies are still necessary, hence we are ready to continue our dialogue and cooperation with all of them.

B.2 Preamble

Informatics, the science of computing, will have an ever growing impact on production, economy, health, science, culture, entertainment, communication and society in general. Several of the innovations we are currently witnessing can be ascribed to the remarkable advances in this discipline, which has gained autonomy as a science with its peculiar ways of thinking, of interpreting the world, as well as of approaching problems.

While we are experiencing a rapid evolution of digital devices and of their applications, the scientific foundations of Informatics are nevertheless firmly rested on a homogeneous range of concepts, methodologies and skills.

In order to cope with the ubiquity of information technology, all citizens must learn the foundations of Informatics and acquire the conceptual tools necessary to understand the logic and the processes underlying the digital world in which they are immersed and on which the quality of their life will depend.

Abstraction, planning and accuracy are essential traits of the problem solving approach in the Informatics field, that foster the development of critical thinking and provide helpful keys to understand complex systems. Whatever their interests and future vocation, all students need to practice these competencies in order to be able to create with, and not merely make use of, the digital technologies.

The instructional program starts from primary school and is organized into three main learning stages.

In the first stage (primary school) students are encouraged to "ask questions," as well as to "discover" in their everyday life and to "explore" some basic ideas of Informatics. They can be engaged either in "plugged," i.e. implying the use of computing devices, or "unplugged" activities, i.e. without using digital technologies, possibly by drawing inspiration from the history of such ideas.

In the second stage (lower secondary school) students are expected to grow in autonomy. To achieve this educational goal, they have to learn more about the organization of data and the concept of algorithm; moreover, they should be offered opportunities to develop abstract thinking and to acquire new specific as well as cross-disciplinary competencies. In particular, programming tasks can play a key role in this respect.

The first two stages lay the foundations for mastering the concepts and for enhancing the competencies at the core of the third stage (upper secondary school), at the end of which students should be able to model problems and to design algorithms.

Whatever the school level, the teaching of Informatics can, by its nature, be approached through active learning methods, teamwork and laboratory activities (including "unplugged" activities). Programming projects, possibly integrating or re-using third-party products, as well as the reasoning to justify program correctness are crucial to the development of creativity, critical thinking, and therefore personal autonomy.

Informatics is often misrepresented as the mere use of digital technologies, but this is a distorted view. By contrast, advances in this knowledge field foster new and meaningful ways to observe, understand and act on the world around us. The general term "computational thinking" is commonly used to refer to these new ways of characterizing natural systems (e.g. living systems) as well as artificial systems (e.g. networks of social relations). To be able to take a thorough "computational thinking" perspective, students need an adequate Informatics education.

Data, information, computation, algorithm, computing machine and formal language are key unifying concepts of the discipline. Although such concepts were already known, to some extent, before the birth of Informatics as an independent field, they have been significantly clarified and deepened precisely because of the major role they play in Informatics.

In addition to its conceptual tools, Informatics provides us with a wide range of methodological tools that enable us to model and to master the complexity of the faced problems faced. This conceptual and methodological baggage is also fundamental for a purposeful and creative use of information technology.

Thus, essential aims of an Informatics curriculum are to develop students' ability: to collect, represent and organize data; to conceive algorithms; to model problems; to reduce the complexity of a problem by breaking it down into smaller parts that are easier to approach; to think at multiple levels of abstraction; to identify recurring patterns; to reuse available solutions for solving similar problems; to describe data, problems and solutions in abstract (artificial) languages.

As Duchâteau [1992] pointed out, Informatics is a relentless endeavor to disclose meaning from form and to confine meaning within form. Moreover, the conceptual understanding of the scope of its tools revealed, according to Mazoyer [2005], the "miracles" that combining a large number of times a small set of elementary operations can achieve a huge potential; that this potential is not specific to some particular type of operations; that the related limits can be expressed and understood formally.

The major goal of the curriculum is to give all students the opportunity to develop basic competences in informatics. In particular, at the end of compulsory school each student should be able:

- to understand and to apply basic concepts and principles of the field;
- to tackle problems by means of the tools and methods of Informatics;
- to analyze problems by devising formal representations, by designing algorithmic solutions and by coding the algorithms in a programming language;
- to evaluate the potential benefits as well as the limits of applying a range of digital technologies to achieve a given task;
- to use digital technologies in a conscious, responsible, confident, competent and creative way.

B.3 Primary School

B.3.1 Competence Goals at the End of Primary School

The student:

T-P-1 understands that an algorithm describes a procedure that can be automated in a precise and unambiguous manner;

T-P-2 understands how an algorithm can be expresses by means of a program written using a programming language;

T-P-3 read and write structurally simple programs;

T-P-4 explain, using logical reasoning, why a structurally simple program achieves its goals or fails;

T-P-5 begins to recognize the difference between information and data

T-P-6 explores the possibility of representing data of various kinds (numbers, images, sounds, ...) using different formats, even arbitrarily chosen ones;

T-P-7 starts recognizing the presence of computers in technological devices of everyday life;

T-P-8 recognizes the Internet as a communication infrastructure, distinguishing it from its services (e.g., search engines, e-mail, WWW) and the contents transmitted;

T-P-9 understands the basic rules for the safe and socially responsible use of information technology;

T-P-10 uses information technology systems to choose and use digital content;

T-P-11 develops a positive attitude towards computer-based applications, recognizing their potential as tools for personal expression in everyday life.

B.3.2 Knowledge and Skills at the End of the Third Grade of Primary School

B.3.2.1 Area of Algorithms

O-P3-A-1 to recognize algorithmic aspects in routine operations of everday life: e.g., brushing one's teeth, dressing, leaving the classroom

O-P3-A-2 to understand that difficult problems can be solved by breaking them down in smaller parts

B.3.2.2 Area of Programming

O-P3-P-1 to notice errors in simple programs and act to correct them;

O-P3-P-2 to order the sequence of instructions correctly;

O-P3-P-3 to use loops to concisely express that a certain action has to be repeatedly executed a prefixed number of times;

O-P3-P-4 to use one-way selection to make decisions within simple programs.

B.3.2.3 Area of Data and Information

O-P3-D-1 to select and use objects to represent data one is familiar with (e.g. colors, words, ...)

O-P3-D-2 to define the interpretation of the objects employed to represent data (i.e., the legend)

B.3.2.4 Area of Digital Awareness

O-P3-N-1 to recognize the uses of informatics and digital technologies in everyday life;

O-P3-N-2 to understand the concept of private data and the need to keep them confidential;

O-P3-N-3 to understand the importance of respecting others when using digital technologies;

O-P3-N-4 to be able to ask for help in case of problems related to downloaded materials or to contacts in which one is involved with on the Internet or through other online technologies.

B.3.2.5 Area of Digital Creativity

O-P3-R-1 to create elementary digital content

O-P3-R-2 to select and to use digital content for expressive purposes, using computer-based applications and digital devices in a simple way.

B.3.3 Knowledge and Skills at the End of the Fifth Grade of Primary School

B.3.3.1 Area of Algorithms

O-P5-A-1 to use logical reasonsing to explain how simple algorithms work

O-P5-A-2 to solve complex problems by breaking them into smaller parts.

B.3.3.2 Area of Programming

O-P5-P-1 to examine the behavior of simple programs to understand and possibily correct them;

O-P5-P-2 to write loops to repeat a certain action while an easy-to-test condition stays true;

O-P5-P-3 to recognize that a sequence of instructions can be considered as a single action, which can be repeated or selected;

O-P5-P-4 to write simple programs that react to events;

O-P5-P-5 to explore the use of two-way selection to implement mutually exclusive actions within simple programs.

B.3.3.3 Area of Data and Information

O-P5-D-1 to use combinations of symbols to represent non-elementary data one is familiar with (e.g. secondary colors, sentences, ...);

O-P5-D-2 to use symbols to represent simple structured data (e.g. bitmap images)

B.3.3.4 Area of Digital Awareness

O-P5-N-1 to know the main hardware and software components of the devices one uses;

O-P5-N-2 to understand the distinction between the communication network and the services accessible through it;

O-P5-N-3 to understand how the privacy of digital data can be protected by "secret" codes;

O-P5-N-4 to recognize acceptable / unacceptable behavior in the use of information technology and of content obtained through it;

O-P5-N-5 to know how to report problems or concerns regarding content obtained or contacts established on the Internet.

B.3.3.5 Area of Digital Creativity

O-P5-R-1 to create simple multimedia content

O-P5-R-2 to create simple computer applications for expressive purposes (eg. stories, games, music, ...) using suitable environments;

O-P5-R-3 to select, modify and combine digital content for expressive purposes, using computer applications and IT in a simple way.

B.4 Lower Secondary School

B.4.1 Competence Goals at the End of Lower Secondary School

The student:

T-M-1 understands the need for precision, so that instructions are always interpreted in the same way by an automatic executor;

T-M-2 algorithmically describes simple processes, such as those encountered in nature and everyday life, or those studied in other disciplines;

T-M-3 understands the importance and the need to reflect on the correctness of the algorithmic descriptions;understands the use of variables to represent data within the program;

T-M-4 designs, writes and debugs, using easy-to-use programming languages, programs that apply selection, loops, variables and elementary forms of input and output;

T-M-5 rearrange programs in order to improve them, by structuring them in modular components as functions and procedures;

T-M-6 recognizes input and output to computer-based applications

T-M-7 understands the different roles of data in a program: input, program state representation, output;

T-M-8 classifies data according to their nature and purpose;

T-M-9 knows the main (physical and functional) architectural principles of a computer-based system;

T-M-10 recognizes the hardware and software components of computer-based systems;

T-M-11 recognizes the fundamental mechanisms by which computer-based systems communicate and provide services on the Internet;

T-M-12 knows the appropriate / inappropriate safe / dangerous ways responsible / irresponsible to use computer-based technologyselects and uses, even in a combined way, computer programs and services to achieve a specific goal

T-M-13 experiments the potential of computer-based applications and digital devices as a tool for personal expression.

B.4.2 Knowledge and Skills at the End of Lower Secondary School

B.4.2.1 Area of Algorithms

O-M-A-1 to detect the potential ambiguities hidden in the description of an algorithm when natural language is used;

O-M-A-2 to describe the algorithms according to the capabilities of the automatic executor and reflects on their correctness;

O-M-A-3 to write algorithms, even based on conventional notations, to describe simple processes inspired from the natural world, from the everyday life, or studied in other disciplines;

O-M-A-4 to detect and describe the conditions under which the above mentioned processes may terminate.

B.4.2.2 Area of Programming

O-M-P-1 to try small/simple changes in a program to understand and modify its behavior, identify and fix its flaws;

O-M-P-2 to write programs that use nested loops and selections;

O-M-P-3 to use in a simple way modular mechanisms, such as functions and procedures

O-M-P-4 to write programs using also typed variables of a simple kind;

O-M-P-5 to use variables that represent the state of the program and allow tracing the progress of computation;

O-M-P-6 to use variables in the conditions of loops and selections;

O-M-P-7 to re-organize programs to improve their comprehensibility.

B.4.2.3 Area of Data and Information

O-M-D-1 to recognize whether two representation of the same simple data are interchangeable for the intended purpose;

O-M-D-2 to perform simple manipulations of symbols that represent structured data (e.g. binary numbers, bitmap images);

O-M-D-3 to use variables to represent the state of a computation;

O-M-D-4 to use structured variables to represent collections of homogeneous data (e.g. vectors, lists, ...).

B.4.2.4 Area of Digital Awareness

O-M-N-1 to understand the main architectural and functional concepts of the Internet and the Web;

O-M-N-2 to understand the main architectural and functional concepts of computer-based systems and devices, distinguishing between hardware and software;

O-M-N-3 to use the most common computer-based systems and devices to organize and manage the data of interest;

O-M-N-4 to connect computer-based devices with each other and with peripheral devices, also with the purpose to realize simple experiences of data collection and analysis and of control of external devices;

O-M-N-5 to recognize the value of any personal data, not only of sensitive data, and be aware of issues related to identity on the network;

O-M-N-6 to understand the social risks connected to the systematic collection of data and the inherently public dimension of social networks;

O-M-N-7 to critically evaluate the content found on the web.

B.4.2.5 Area of Digital Creativity

O-M-R-1 to experiment during the creation of digital content various digital tools and multiple processing methods, so as to express themselves at their best;

O-M-R-2 to choose the most appropriate digital tools for their expressive goals;

O-M-R-3 to create software applications for expressive purposes (eg. stories, games, music, ...) using suitable environments;

O-M-R-4 to select and organize digital content for an effective presentation.

B.5 First Biennium of Higher Secondary School

B.5.1 Competence Goals at the End of the First Biennium of Higher Secondary School

The student:

T-S-1 understands the need to refer to the capabilities of an automatic executor in order to express algorithms in an unambigous way;

T-S-2 recognizes that algorithms are able to solve problems in their generality, and not for single instances;

T-S-3 is able to discuss the correctness of an algorithm with respect to the above mentioned generality;

T-S-4 understands the nature of the problems that are worth an algorithmic solution;

T-S-5 is able to evaluate the efficiency of simple algorithms;

T-S-6 defines, implements and validates programs and systems that model or simulate simple physical systems or familiar processes that occur in the real world or are studied in other disciplines;

T-S-7 understands when programming can provide a convenient way of tackling a problem;

T-S-8 understands the conventional nature of the representation chosen for the data, according to the described information,

T-S-9 recognizes that the way data are represented and organized affects the effectiveness and the efficiency of computation

T-S-10 selects and recognizes in computer programs the representations of problem data, obtained results and elements tracing the intermediate state of a computation;

T-S-11 recognizes the universal and multi-purpose nature of computer-based systems and understands the role of programs in transforming them into machines for specific purposes;

T-S-12 understands the importance of user needs for the implementation of computer-based applications;

T-S-13 is aware that the Internet and computer-based systems and devices influence the economy and the organization of society;

T-S-14 is aware that the diffusion and use of information technology has ethical and social consequences and learns to evaluate them critically;

T-S-15 selects, uses and combines computer programs and software services to develop well-structured informatics projects;

T-S-16 selects, combines and extends computational artifacts to express its own creativity

B.5.2 Knowledge and Skills at the End of the First Biennium Of Higher Secondary School

B.5.2.1 Area of Algorithms

O-S-A-1 to know a selection of simple algorithms that solve fundamental problems (for example, search, sorting, maximum common divisor, etc.);

O-S-A-2 to use logical reasoning to evaluate different algorithms that solve the same problem;

O-S-A-3 to understand that not all problems can be solved algorithmically;

O-S-A-4 to take into account, when designing an algorithm, the characteristics of the automatic executor and the limits of its resources;

B.5.2.2 Area of Programming

O-S-P-1 to recognize how the various parts of a program contribute to its functioning;

O-S-P-2 to predict the result of a program without running it;

O-S-P-3 to use conditions that use a logical operator;

O-S-P-4 to use loops with conditions to describe parametric actions;

O-S-P-5 to write programs using structured variables;

O-S-P-6 to design and develop modular programs using procedures and functions;

O-S-P-7 to write simple programs in a textual programming language, respecting their syntax;

B.5.2.3 Area of Data and Information

O-S-D-1 to evaluate the advantages and disadvantages of alternative representations of the same data

O-S-D-2 to know the characteristics of the fundamental data structures (eg: lists, vectors, matrices, dictionaries, ...) and to know how to select the most suitable one to accomplish the task at hand;

O-S-D-3 to recognize the difference between data and metadata in some simple context (e.g., HTML, simple data description languages, ...).

B.5.2.4 Area of Digital Awareness

O-S-N-1 to realize experiences of data collection and analysis through sensors and of control of external devices;

O-S-N-2 to take into account the requirements of end-users in the implementation of computer-based applications;

O-S-N-3 to identify if and how digital programs and contents can be reused, modified, disseminated;

O-S-N-4 to be aware of the multifaceted relations between the protection of the privacy of individual data and the protection of company security (e.g., anonymity in the network, ...);

O-S-N-5 to evaluate the reliability of content found on the web, examining and double checking sources

B.5.2.5 Area of Digital Creativity

O-S-R-1 to use programming environments for expressive purposes (eg animations, sound tracks, games, ...);

O-S-R-2 to combine programming and online services to achieve its own goals

Appendix C

Questionnaires for mindset related measures in High School

In this appendix the questionnaires used for the study about mindset in high school (described in Chapter 12 and published in Lodi [2019]) are reported.

Here is reported the English translation of the questionnaires used (that are in Italian).
Meta-comments are in italics/square brackets, not part of the text of the questionnaire.

C.1 Pre-Post Questionnaires

The pre and post questionnaires for CS and non-CS classes have a very similar structure. Differences are highlighted.
*Question marked with * are reverse scored so that a low score always indicates a less desirable belief (e.g. fixed mindset), and a high score a more desirable belief (e.g. growth mindset).*
Questions were presented in random order within each section.

General Questions

- Gender
- Before this school year, you had already programmed?
- *[Shown only if answered 'yes' to the previous question]* In which programming languages you had already programmed?
- *[Only in post-test for CS-oriented]* With respect to the first term, your grades in CS related disciplines... (improved [+1] / remained the same [0] / worsened [-1]).

General Beliefs

[These questions are based on Dweck [2003], an italian translation of the original Dweck [1999].]

Growth mindset

Rate from 1 (totally disagree) to 6 (totally agree).

- * You have a certain amount of intelligence, and you can't really do much to change it.
- * Your intelligence is something about you that you can't change very much.
- * You can learn new things, but you can't really change your basic intelligence.

Confidence in one's intelligence

Choose the most true between two sentences, then show how true is it for you, from 1 (sort of true for me) to 6 (very true for me).
[Scores were re-scaled from 12 (very true, high confidence) to 1 (sort of true, low confidence)]

- I usually think I'm intelligent / I wonder if I'm intelligent
- When I get new work in school, I'm usually sure I will be able to learn it. / When I get new work in school, I often think I may not be able to learn it
- I'm not very confident about my intellectual ability / I feel pretty confident about my intellectual ability

Task choice goal measure

Rate from 1 (totally disagree) to 6 (totally agree).

- * If I knew I wasn't going to do well at a task, I probably wouldn't do it even if I might learn a lot from it
- * Although I hate to admit it, I sometimes would rather do well in a class of than learn a lot
- It's much more important for me to learn things in my classes than it is to get the best grades

Choose one of the two options.

- If I had to choose between getting good grades and being challenged in class, I would choose... (Good grades [1] / Being challenged [6])

CS Beliefs

[These scales were adapted from Sun [2015], a thesis about Mathematical Mindsets, changing the word "Math" with "Computer Science".]

CS Mindset

Rate from 1 (totally disagree) to 6 (totally agree).

- Anyone can be good at computer science if the word hard at it
- * You have a certain amount of "computer science intelligence", and you can't really do much to change it.
- * There are limits to how much people can improve their basic CS ability.
- * Some students can never do well in computer science, even if they try hard.

Nature of CS

Rate from 1 (totally disagree) to 6 (totally agree).

- * There is usually only one way to solve a CS problem
- * CS involves mostly facts and procedures that have to be learned.
- * People who really understand CS will have a solution quickly.
- Mistakes are important when learning CS

Identification with CS

[For non-CS oriented classes, "is" and "will be" were changed in "would be".]

- I see myself as a "CS person". (from 1 – totally disagree to 5 – totally agree)
- I think CS is (easy [5] / hard [1]) for me.
- I will be (bad [1] / good [5]) at CS.

CS Mastery orientation

Rate from 1 (not at all important) to 5 (extremely important).
[For non-CS oriented classes, second and fourth question was rephrased starting with "If you studied CS, . . . ".]

- * In CS, how important is it to avoid making mistakes?
- * How important is it that you to do better than other students in your CS class?
- * In CS class, how important is it to get the right answer?
- * How important is it that you do well in CS?

C.2 Open Questions for the Intervention Class

In class INF1, halfway through the year, we performed a mindset intervention. At the end of the lesson, students were asked to reflect on struggle and answer the following open-ended questions.

Reflections after the Lesson

Reflect on a situation when you struggled to learn something. It could be anything – learning a new math concept, or a new technique in soccer, or staring at the blank page during the writing of an essay. A situation where you failed at first but
through persevering, hard work, others help and using different strategies, you succeeded or you became better at the task at hand. How do you felt? How did you overcome that situation and what did you learn from it? *[Adapted from Khan Academy and PERTS [2014]]*

1. Letter to mates

Write a letter to a future student of your class about this struggle. In at least five sentences, tell this student your story and give him advice on what he should do next time he encounters an obstacle when learning something new. Feel free to be as creative as you would like, but try to write a useful letter. *[Adapted from Khan Academy and PERTS [2014]]*

2. What should a teacher do to stimulate a "growth mindset" in students?

Feel completely free to express what you think would be the best for yourself, and explain why.

3. Suggestions to mates about programming

From September 2017, or perhaps from before, you started to program. Maybe it was very easy for you, or on the contrary, you encountered many difficulties. Computer errors happen all the time, and even the most experienced programmers spend a lot of time debugging, that is, hunting for errors. Write in at least five sentences some tips that you would give to a classmate who has to start learning to program, recommending him **concrete strategies** to succeed and not to be discouraged in the face of difficulties related to learning programming. *[Inspired by the intervention from Simon et al. [2008]]*

Appendix D

Questionannaries for GM and Computer Anxiety in Pre-service teachers

In this appendix the questionnaires used for the study about mindset and computer anxiety (described in Chapter 13 and published in Lodi [2018a]) are reported.

D.1 Questionnaires

D.1.1 Implicit Theories of Intelligence Scale

It included eight Likert-type (1 to 6) items, described in Dweck [1999, p. 285]. We used an Italian translation. Questions with * indicate a fixed mindset, so they were reverse scored (so that high agreement corresponds with a growth mindset for all questions)

Q1* You have a certain amount of intelligence, and you can't really do much to change it.

Q2* Your intelligence is something about you that you can't change very much.

Q3 No matter who you are, you can significantly change your intelligence level.

Q4* To be honest, you can't really change how intelligent you are.

Q5 You can always substantially change how intelligent you are.

Q6* You can learn new things, but you can't really change your basic intelligence.

Q7 No matter how much intelligence you have, you can always change it quite a bit.

Q8 You can change even your basic intelligence level considerably.

D.1.2 Computer Anxiety Rating Scale

It included nineteen Likert-type (1 to 5) items, described in Heinssen et al. [1987] (we used an Italian translation of the slight variation proposed by Sam et al. [2005]). Questions with * indicate a low level of anxiety, so they were reverse scored (so that high agreement corresponds with high anxiety for all questions)

Q1 I feel insecure about my ability to interpret a computer printout

Q2* I look forward to using a computer on my job

Q3 I do not think I would be able to learn a computer programming language

Q4* The challenge of learning about computers is exciting

Q5* I am confident that I can learn computer skills

Q6* Anyone can learn to use a computer is they are patient and motivated

Q7* Learning to operate computers is like learning any new skill, the more you practice, the better you become

Q8 I am afraid that if I begin to use computer more, I will become more dependent upon them and lose some of my reasoning skills

Q9* I am sure that with time and practice I will be as comfortable working with computers as I am in working by hand

Q10* I feel that I will be able to keep up with the advances happening in the computer field

Q11 I would dislike working with machines that are smarter than I am

Q12 I feel apprehensive about using computers

Q13 I have difficulty in understanding the technical aspects of computers

Q14 It scares me to think that I could cause the computer to destroy a large amount of information by hitting the wrong key

Q15 I hesitate to use a computer for fear of making mistakes that I cannot correct

Q16 You have to be a genius to understand all the special keys contained on most computer terminals

Q17* If given the opportunity, I would like to learn more about and use computers more

Q18 I have avoided computers because they are unfamiliar and somewhat intimidating to me

Q19* I feel computers are necessary tools in both educational and work settings

Bibliography

Harold Abelson et al. 1998. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* 11, 1 (1 Aug. 1998), 7–105. https://doi.org/10.1023/A:1010051815785 [Cited on page 108]

Harold Abelson and Andrea A. diSessa. 1981. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press, Cambridge, MA, USA. [Cited on page 105]

Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs* (second ed.). MIT Press, Cambridge, MA, USA. [Cited on pages 101 and 108]

Alireza Ahadi and Raymond Lister. 2013. Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant?. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) *(ICER '13)*. Association for Computing Machinery, New York, NY, USA, 123–128. https://doi.org/10.1145/2493394.2493416 [Cited on pages 4, 63, and 183]

Alfred V. Aho. 2011. Ubiquity Symposium: Computation and Computational Thinking. *Ubiquity* 2011, January, Article 1 (2011), 8 pages. https://doi.org/10.1145/1922681.1922682 [Cited on pages 29 and 82]

Eric Allen, Robert Cartwright, and Brian Stoler. 2002. DrJava: A Lightweight Pedagogic Environment for Java. *SIGCSE Bull.* 34, 1 (Feb. 2002), 137–141. https://doi.org/10.1145/563517.563395 [Cited on page 108]

Susan A. Ambrose, Michael W. Bridges, Michele DiPietro, Marsha C. Lovett, and Marie K. Norman. 2010. *How learning works: Seven research-based principles for smart teaching*. John Wiley & Sons. [Cited on pages 4, 51, 55, and 56]

Katerina Ananiadou and Magdalena Claro. 2009. *21st Century Skills and Competences for New Millennium Learners in OECD Countries*. OECD Education Working Papers 41. OECD Publishing, Paris. [Cited on page 79]

Mikko-Ville Apiola and Mikko-Jussi Laakso. 2019. The Impact of Self-Theories to Academic Achievement and Soft Skills in Undergraduate CS Studies: First Findings. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*

(Aberdeen, Scotland, UK) *(ITiCSE '19)*. ACM, New York, NY, USA, 16–22. https://doi.org/10.1145/3304221.3319766 [Cited on page 67]

Barbara Arfé, Tullio Vardanega, and Lucia Ronconi. 2020. The effects of coding on children's planning and inhibition skills. *Computers & Education* 148 (April 2020), 103807. https://doi.org/10.1016/j.compedu.2020.103807 [Cited on page 60]

Michal Armoni. 2016. Computing in Schools: Computer Science, Computational Thinking, Programming, Coding: The Anomalies of Transitivity in K-12 Computer Science Education. *ACM Inroads* 7, 4 (Nov. 2016), 24–27. https://doi.org/10.1145/3011071 [Cited on pages 34, 35, 36, and 81]

John W. Backus. 1959. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In *Proceedings Int. Conf. on Information Processing, UNESCO*. 125–132. [Cited on page 72]

John W. Backus et al. 1960. Report on the Algorithmic Language ALGOL 60. *Commun. ACM* 3, 5 (May 1960), 299–314. https://doi.org/10.1145/367236.367262 [Cited on page 72]

Štěpán Bahník and Marek A. Vranka. 2017. Growth mindset is not associated with scholastic aptitude in a large sample of university applicants. *Personality and Individual Differences* 117 (Oct. 2017), 139–143. https://doi.org/10.1016/j.paid.2017.05.046 [Cited on page 65]

Lindsay Baker, Stella Ng, and Farah Friesen. 2019. *Paradigms of Education. An Online Supplement.* Retrieved February 1, 2020 from https://www.paradigmsofeducation.com [Cited on pages 39, 40, 41, and 44]

Erik Barendsen, Linda Mannila, Barbara Demo, Nataša Grgurina, Cruz Izu, Claudio Mirolo, Sue Sentance, Amber Settle, and Gabrielė Stupurienė. 2015. Concepts in K-9 Computer Science Education. In *Proceedings of the 2015 ITiCSE on Working Group Reports* (Vilnius, Lithuania) *(ITICSE-WGR '15)*. Association for Computing Machinery, New York, NY, USA, 85–116. https://doi.org/10.1145/2858796.2858800 [Cited on pages 20 and 35]

Susan M. Barnett and Stephen J. Ceci. 2002. When and where do we apply what we learn?: A taxonomy for far transfer. *Psychological Bulletin* 128, 4 (2002), 612–637. https://doi.org/10.1037/0033-2909.128.4.612 [Cited on page 52]

Ashok Basawapatna, Kyu Han Koh, Alexander Repenning, David C. Webb, and Krista Sekeres Marshall. 2011. Recognizing Computational Thinking Patterns. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. ACM, New York, NY, USA, 245–250. https://doi.org/10.1145/1953163.1953241 [Cited on page 57]

Walter F. Bauer. 2007. Computer Recollections: Events, Humor, and Happenings. *IEEE Annals of the History of Computing* 29, 1 (Jan. 2007), 85–89. https://doi.org/10.1109/MAHC.2007.2 [Cited on page 1]

Kent Beck and Cynthia Andres. 2004. *Extreme Programming Explained: Embrace Change* (second ed.). Addison-Wesley Professional. [Cited on page 122]

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. 2001. Manifesto for Agile Software Development. http://agilemanifesto.org/iso/en/manifesto.html. [Cited on pages 121 and 122]

Tim Bell. 2016. What's all the fuss about coding?. In *2009 - 2019 ACER Research Conferences (ACER Research Conference 2016)*. Australian Council for Educational Research (ACER), Melbourne, Vic, Australia, 12–16. https://research.acer.edu.au/research_conference/RC2016/8august/12 [Cited on pages 34 and 36]

Tim Bell and Michael Lodi. 2019a. Constructing Computational Thinking Without Using Computers. *Constructivist Foundations* 14, 3 (2019), 342–351. https://constructivist.info/14/3/342.bell [Cited on pages ix, 10, and 97]

Tim Bell and Michael Lodi. 2019b. Authors' Response: Keeping the "Computation" in "Computational Thinking" Through Unplugged Activities. *Constructivist Foundations* 14, 3 (2019), 357–359. https://constructivist.info/14/3/357.bell [Cited on pages ix, 10, and 97]

Tim Bell, Frances Rosamond, and Nancy Casey. 2012. Computer Science Unplugged and Related Projects in Math and Computer Science Popularization. In *The Multivariate Algorithmic Revolution and Beyond: Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday*, Hans L. Bodlaender, Rod Downey, Fedor V. Fomin, and Dániel Marx (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 398–456. https://doi.org/10.1007/978-3-642-30891-8_18 [Cited on page 116]

Tim Bell, Paul Tymann, and Amiram Yehudai. 2018. The Big Ideas in Computer Science for K-12 Curricula. *Bulletin of EATCS* 1, 124 (2018). [Cited on pages 17 and 120]

Tim Bell and Jan Vahrenhold. 2018. CS Unplugged—How Is It Used, and Does It Work? In *Adventures Between Lower Bounds and Higher Altitudes: Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*, Hans-Joachim Böckenhauer, Dennis Komm, and Walter Unger (Eds.). Springer International Publishing, Cham, 497–521. https://doi.org/10.1007/978-3-319-98355-4_29 [Cited on page 117]

Carlo Bellettini, Violetta Lonati, Dario Malchiodi, Mattia Monga, Anna Morpurgo, and Federico Pedersini. 2015. La formazione degli insegnanti della classe 42/A–Informatica: l'esperienza dell'Università degli Studi di Milano. In *E questo tutti chiamano informatica: L'esperienza dei TFA nelle discipline informatiche*. Collana Manuali, Vol. 14. Sapienza Univ. Ed., Chapter 4, 53–76. In Italian. [Cited on page 94]

Carlo Bellettini, Violetta Lonati, Dario Malchiodi, Mattia Monga, Anna Morpurgo, Mauro Torelli, and Luisa Zecca. 2014. Informatics Education in Italian Secondary Schools. *ACM*

Trans. Comput. Educ. 14, 2, Article 15 (June 2014), 6 pages. https://doi.org/10.1145/2602490 [Cited on page 22]

Mordechai Ben-Ari. 2001. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching* 20, 1 (2001), 45–73. [Cited on page 102]

Mordechai Ben-Ari. 2015. In Defense of Programming. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (Vilnius, Lithuania) *(ITiCSE '15)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2729094.2742581 [Cited on page 34]

Elena Bender, Peter Hubwieser, Niclas Schaper, Melanie Margaritis, Marc Berges, Laura Ohrndorf, Johannes Magenheim, and Sigrid Schubert. 2015. Towards a Competency Model for Teaching Computer Science. *Peabody Journal of Education* 90, 4 (2015), 519–532. https://doi.org/10.1080/0161956X.2015.1068082 [Cited on page 2]

Walter Bender. 2017. La plataforma de aprendizaje Sugar: Affordances educativas para el pensamiento computacional. *Revista de Educación a Distancia* 17, 54 (Jul. 2017). https://revistas.um.es/red/article/view/298791 [Cited on page 75]

Miles Berry. 2015. *Teaching computing*. Retrieved February 1, 2020 from http://milesberry.net/2015/02/teaching-computing/ [Cited on page 174]

Michael Berry and Michael Kölling. 2014. The State of Play: A Notional Machine for Learning Programming. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education* (Uppsala, Sweden) *(ITiCSE '14)*. ACM, New York, NY, USA, 21–26. https://doi.org/10.1145/2591708.2591721 [Cited on page 101]

Lisa S. Blackwell, Kali H. Trzesniewski, and Carol S. Dweck. 2007. Implicit Theories of Intelligence Predict Achievement Across an Adolescent Transition: A Longitudinal Study and an Intervention. *Child Development* 78, 1 (Jan. 2007), 246–263. https://doi.org/10.1111/j.1467-8624.2007.00995.x [Cited on pages 64 and 177]

Paulo Blikstein. 2018. *Pre-College Computer Science Education: A Survey of the Field.* Technical Report. Mountain View, CA: Google LLC. https://goo.gl/gmS1Vm [Cited on page 58]

Paulo Blikstein and Sepi Hejazi Moghadam. 2019. Computing Education Literature Review and Voices from the Field. In *The Cambridge Handbook of Computing Education Research*. Cambridge University Press, 56–78. https://doi.org/10.1017/9781108654555.004 [Cited on page 58]

Jo Boaler. 2013. Ability and Mathematics: the mindset revolution that is reshaping education. *FORUM* 55, 1 (2013), 143–152. https://doi.org/10.2304/forum.2013.55.1.143 [Cited on pages 65 and 66]

Jo Boaler. 2015. *Mathematical mindsets: Unleashing students' potential through creative math, inspiring messages and innovative teaching*. John Wiley & Sons. [Cited on page 65]

Roberto Borchia, Antonella Carbonaro, Giorgio Casadei, Luca Forlizzi, Michael Lodi, and Simone Martini. 2018. Problem Solving Olympics: An Inclusive Education Model for Learning Informatics. In *Informatics in Schools. Fundamentals of Computer Science and Software Engineering. Lecture Notes in Computer Science (ISSEP 2018)*, Sergei N. Pozdniakov and Valentina Dagienė (Eds.), Vol. 11169. Springer International Publishing, Cham, 319–335. [Cited on pages x and 20]

Pierre Bourdieu. 1977. *Outline of a Theory of Practice*. Cambridge University Press, Cambridge. [Cited on page 79]

Matt Bower and Katrina Falkner. 2015. Computational Thinking, the Notional Machine, Pre-service Teachers, and Research Opportunities. In *Proceedings of the 17th Australasian Computing Education Conference* (Sydney, NSW, Australia) *(ACE 2015)*. Australian Computer Society Inc., Sydney, NSW, Australia, 37–46. [Cited on pages 147 and 153]

Russell Boyatt, Meurig Beynon, and Megan Beynon. 2014. Ghosts of Programming Past, Present and Yet to Come. In *Proc. of the 25th Annual Workshop of the Psychology of Programming Interest Group* (Brighton, UK) *(PPIG 2014)*, Benedict du Boulay and Judith Good (Eds.). 171–182. [Cited on page 19]

Laura Branchetti, Olivia Levrini, Eleonora Barelli, Michael Lodi, Giovanni Ravaioli, Laura Rigotti, Sara Satanassi, and Giulia Tasquier. 2019. STEM analysis of a module on Artificial Intelligence for high school students designed within the I SEE Erasmus+ Project. In *Proceedings of the Eleventh Congress of the European Society for Research in Mathematics Education (CERME11)*, Uffe Thomas Jankvist, Marja van den Heuvel-Panhuizen, and Michiel Veldhuis (Eds.), Vol. TWG26. Utrecht University, Freudenthal Group, Utrecht, Netherlands. https://hal.archives-ouvertes.fr/hal-02410332 [Cited on page xi]

Karen Brennan, Christan Balch, and Michelle Chung. 2014. *Scratch Curriculum Guide*. Retrieved February 1, 2020 from http://scratched.gse.harvard.edu/guide/ [Cited on pages 111, 128, 187, and 188]

Karen Brennan and Mitchel Resnick. 2012. New frameworks for studying and assessing the development of computational thinking (Using artifact-based interviews to study the development of computational thinking in interactive media design). In *Proceedings of the 2012 annual meeting of the American Educational Research Association* (Vancouver, Canada) *(AREA 2012)*. 25 pages. http://scratched.gse.harvard.edu/ct/files/AERA2012.pdf [Cited on pages 28, 30, 31, and 32]

David Brin. 2016. *Why Johnny can't code*. Retrieved February 1, 2020 from https://www.salon.com/2006/09/14/basic_2/ [Cited on page 107]

Ann L. Brown. 1992. Design Experiments: Theoretical and Methodological Challenges in Creating Complex Interventions in Classroom Settings. *Journal of the Learning Sciences* 2, 2 (April 1992), 141–178. https://doi.org/10.1207/s15327809jls0202_2 [Cited on page 54]

BT and Ipsos-MORI. 2016. *Tech Literacy: A New Cornerstone of Modern Primary School Education.* Technical Report. BT. https://www.btplc.com/Digitalimpactandsustainability/Buildingbetterdigitallives/Techliteracy/IPSOS-full.pdf [Cited on page 59]

Quinn Burke. 2012. The Markings of a New Pencil: Introducing Programming-as-Writing in the Middle School Classroom. *Journal of Media Literacy Education* 4, 2 (2012), 121–135. https://digitalcommons.uri.edu/jmle/vol4/iss2/3 [Cited on page 19]

Alex B. Cannara. 1976. *Experiments in Teaching Children Computer Programming*. Ph.D. Dissertation. School of Education, Stanford University. [Cited on page 58]

Michael E. Caspersen, Judith Gal-Ezer, Andrew McGettrick, and Enrico Nardelli. 2018. *Informatics for All The Strategy*. Technical Report. ACM Europe & Informatics Europe. https://www.acm.org/binaries/content/assets/public-policy/acm-europe-ie-i4all-strategy-2018.pdf [Cited on page 20]

Stephen J. Ceci. 1991. How much does schooling influence general intelligence and its cognitive components? A reassessment of the evidence. *Developmental Psychology* 27, 5 (Sept. 1991), 703–722. https://doi.org/10.1037/0012-1649.27.5.703 [Cited on page 54]

Jeanne Century, Kaitlyn Ferris, and Huifang Zuo. 2018. *Finding Time for Computer Science in the Elementary Day. Preliminary Findings of an Exploratory Study.* Technical Report. Outlier Research & Evaluation - UChicago STEM Education at the University of Chicago. https://s3.amazonaws.com/cemse/time-for-cs/docs/TimeforCS_Preliminary_Findings.pdf [Cited on page 59]

Michael Clancy. 2004. Misconceptions and attitudes that interfere with learning to program. In *Computer science education research*, Sally Fincher and Marian Petre (Eds.). Routledge, 85–100. [Cited on page 102]

College Board. 2017. *AP Computer Science Principles.* Technical Report. College Board. https://apcentral.collegeboard.org/pdf/ap-computer-science-principles-course-and-exam-description.pdf [Cited on page 208]

Committee on Science Education. 2013. *Teaching computer science in France: Tomorrow can't wait*. Technical Report. Institut de France – Académie des Sciences. http://www.academie-sciences.fr/en/Advice-Notes-and-Reports/teaching-computer-science-in-france-tomorrow-can-t-wait.html [Cited on pages 19, 81, and 88]

Computing at School. 2012. *Computer Science: A Curriculum for Schools*. Technical Report. Computing at School. http://www.computingatschool.org.uk/data/uploads/ComputingCurric.pdf [Cited on page 19]

Isabella Corradini, Michael Lodi, and Enrico Nardelli. 2017a. Computational Thinking in Italian Schools: Quantitative Data and Teachers' Sentiment Analysis after Two Years of

"Programma Il Futuro". In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 224–229. https://doi.org/10.1145/3059009.3059040 [Cited on pages ix, 11, 17, 24, and 137]

Isabella Corradini, Michael Lodi, and Enrico Nardelli. 2017b. Conceptions and Misconceptions about Computational Thinking among Italian Primary School Teachers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 136–144. https://doi.org/10.1145/3105726.3106194 [Cited on pages ix, 7, 11, 28, and 145]

Isabella Corradini, Michael Lodi, and Enrico Nardelli. 2018a. Coding and Programming: What Do Italian Primary School Teachers Think? (Abstract Only). In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 1074. https://doi.org/10.1145/3159450.3162268 [Cited on pages xi and 11]

Isabella Corradini, Michael Lodi, and Enrico Nardelli. 2018b. An Investigation of Italian Primary School Teachers' View on Coding and Programming. In *Informatics in Schools. Fundamentals of Computer Science and Software Engineering. Lecture Notes in Computer Science (ISSEP 2018)*, Sergei N. Pozdniakov and Valentina Dagienė (Eds.), Vol. 11169. Springer International Publishing, Cham, 228–243. [Cited on pages x, 11, 17, 34, and 145]

Andrew Csizmadia, Paul Curzon, Mark Dorling, Simon Humphreys, Thomas Ng, Cynthia Selby, and John Woollard. 2015. *Computational thinking - a guide for teachers*. Technical Report. Computing at School. https://eprints.soton.ac.uk/424545/ [Cited on pages 28, 30, 31, and 32]

CSTA. 2017. *CSTA K-12 Computer Science Standards, Revised 2017*. Technical Report. Computer Science Teachers Association. http://www.csteachers.org/standards [Cited on page 19]

Paul Curzon, Peter W. McOwan, Nicola Plant, and Laura R. Meagher. 2014. Introducing Teachers to Computational Thinking Using Unplugged Storytelling. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (Berlin, Germany) *(WiPSCE '14)*. ACM, New York, NY, USA, 89–92. https://doi.org/10.1145/2670757.2670767 [Cited on page 118]

Quintin Cutts, Emily Cutts, Stephen Draper, Patrick O'Donnell, and Peter Saffrey. 2010. Manipulating Mindset to Positively Influence Introductory Programming Performance. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10)*. Association for Computing Machinery, New York, NY, USA, 431–435. https://doi.org/10.1145/1734263.1734409 [Cited on pages 66 and 67]

Quintin Cutts, Sarah Esper, and Beth Simon. 2011. Computing As the 4th "R": A General Education Approach to Computing Education. In *Proceedings of the Seventh International*

Workshop on Computing Education Research (Providence, Rhode Island, USA) *(ICER '11)*. ACM, New York, NY, USA, 133–138. https://doi.org/10.1145/2016911.2016938 [Cited on page 60]

Valentina Dagienė. 2018. Resurgence of Informatics Education in Schools. In *Adventures Between Lower Bounds and Higher Altitudes: Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*, Hans-Joachim Böckenhauer, Dennis Komm, and Walter Unger (Eds.). Springer International Publishing, Cham, 522–537. https://doi.org/10.1007/978-3-319-98355-4_30 [Cited on page 20]

Valentina Dagienė and Sue Sentance. 2016. It's Computational Thinking! Bebras Tasks in the Curriculum. In *Informatics in Schools: Improvement of Informatics Knowledge and Perception*, Andrej Brodnik and Françoise Tort (Eds.). Springer International Publishing, Cham, 28–39. [Cited on page 20]

Wanda P. Dann, Stephen Cooper, and Randy Pausch. 2008. *Learning to program with Alice*. Prentice Hall Press. [Cited on page 111]

Rachel Davies and Liz Sedley. 2009. *Agile Coaching*. The Pragmatic Bookshelf. [Cited on page 121]

Renzo Davoli, Michael Lodi, and Rebecca Montanari. 2017a. *Domo: un maggiordomo "computazionale"*. Retrieved February 1, 2020 from http://www.cs.unibo.it/~renzo/DIDATTICA/DomoCompleto.pdf In Italian. [Cited on page 126]

Renzo Davoli, Michael Lodi, and Rebecca Montanari. 2017b. *Domo: un maggiordomo "computazionale" - Materiali stampabili*. Retrieved February 1, 2020 from http://www.cs.unibo.it/~renzo/DIDATTICA/domo/ In Italian. [Cited on page 126]

Diana DeMarco-Brown. 2013. *Agile User Experience Design - A Practitioner's Guide to making it work*. Elsevier. https://doi.org/10.1016/c2011-0-06229-4 [Cited on page 121]

Peter J. Denning. 2009. The Profession of IT: Beyond Computational Thinking. *Commun. ACM* 52, 6 (June 2009), 28–30. https://doi.org/10.1145/1516046.1516054 [Cited on pages 18 and 72]

Peter J. Denning. 2013. The science in computer science. *Commun. ACM* 56, 5 (May 2013), 35. https://doi.org/10.1145/2447976.2447988 [Cited on page 2]

Peter J. Denning. 2017. Remaining Trouble Spots with Computational Thinking. *Commun. ACM* 60, 6 (May 2017), 33–39. https://doi.org/10.1145/2998438 [Cited on pages 33 and 72]

Peter J. Denning, D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. 1989. Computing as a discipline. *Commun. ACM* 32, 1 (Feb. 1989), 9–23. https://doi.org/10.1145/63238.63239 [Cited on page 1]

Peter J. Denning and Paul S. Rosenbloom. 2009. Computing: The Fourth Great Domain of Science. *Commun. ACM* 52, 9 (Sept. 2009), 27. https://doi.org/10.1145/1562164.1562176 [Cited on page 18]

Peter J. Denning and Matti Tedre. 2019. *Computational Thinking*. MIT Press. [Cited on pages 72 and 209]

Peter J. Denning, Matti Tedre, and Pat Yongpradit. 2017. Misconceptions about Computer Science. *Commun. ACM* 60, 3 (Feb. 2017), 31–33. https://doi.org/10.1145/3041047 [Cited on pages 33, 36, 72, and 146]

Department for Education. 2013. *National Curriculum for England: Computing programme of study*. Technical Report. Department for Education. https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study [Cited on pages 88 and 89]

Edsger W. Dijkstra. 1974. Programming as a Discipline of Mathematical Nature. *The American Mathematical Monthly* 81, 6 (1974), 608–612. https://doi.org/10.1080/00029890.1974.11993624 [Cited on page 73]

Edsger W. Dijkstra. 1985. On anthropomorphism in science. EWD936. https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD936.PDF. [Cited on page 114]

Andrea A. diSessa. 1985. A Principled Design for an Integrated Computational Environment. *Human–Computer Interaction* 1, 1 (1985), 1–47. https://doi.org/10.1207/s15327051hci0101_1 [Cited on page 107]

Andrea A. diSessa. 2000. *Changing Minds: Computers, Learning, and Literacy*. MIT Press, Cambridge, MA, USA. [Cited on page 107]

Andrea A. diSessa. 2018. Computational Literacy and "The Big Picture" Concerning Computers in Mathematics Education. *Mathematical Thinking and Learning* 20, 1 (2018), 3–31. https://doi.org/10.1080/10986065.2018.1403544 [Cited on pages 33, 54, 60, 61, and 107]

Andrea A. diSessa and Harold Abelson. 1986. Boxer: A Reconstructible Computational Medium. *Commun. ACM* 29, 9 (Sept. 1986), 859–868. https://doi.org/10.1145/6592.6595 [Cited on page 107]

Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9 [Cited on pages 100 and 101]

Charles Duchâteau. 1992. Peut-on définir une "culture informatique"? *Journal de Réflexion sur l'Informatique* 23-24 (1992), 34–39. [Cited on page 213]

Caitlin Duncan. 2019. *Computer science and computational thinking in primary schools*. Ph.D. Dissertation. University of Canterbury. http://hdl.handle.net/10092/17160 [Cited on pages 33 and 60]

Caitlin Duncan, Tim Bell, and James Atlas. 2017. What Do the Teachers Think?: Introducing Computational Thinking in the Primary School Curriculum. In *Proceedings of the Nineteenth Australasian Computing Education Conference* (Geelong, VIC, Australia) *(ACE '17)*. ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/3013499.3013506 [Cited on page 147]

Caitlin Duncan, Tim Bell, and Steve Tanimoto. 2014. Should Your 8-Year-Old Learn Coding?. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (Berlin, Germany) *(WiPSCE '14)*. Association for Computing Machinery, New York, NY, USA, 60–69. https://doi.org/10.1145/2670757.2670774 [Cited on pages 34 and 35]

Carol S. Dweck. 1999. *Self-theories: Their role in motivation, personality, and development*. Psychology Press. [Cited on pages 63, 66, 176, 190, 223, and 227]

Carol S. Dweck. 2003. *Teorie del Sé. Intelligenza, motivazione, personalità e sviluppo*. Erickson. In Italian. [Cited on pages 176 and 223]

Carol S. Dweck. 2008. *Mindsets and Math/Science Achievement*. Technical Report. The Opportunity Equation. [Cited on page 65]

Carol S. Dweck. 2017a. *Growth mindset is on a firm foundation, but we're still building the house*. Retrieved February 1, 2020 from https://mindsetscholarsnetwork.org/growth-mindset-firm-foundation-still-building-house/ [Cited on page 65]

Carol S. Dweck. 2017b. *Mindset (Updated Edition)*. Robinson. [Cited on pages 4, 63, 64, 65, 66, and 67]

Carol S. Dweck, Chi-yue Chiu, and Ying-yi Hong. 1995. Implicit Theories and Their Role in Judgments and Reactions: A Word From Two Perspectives. *Psychological Inquiry* 6, 4 (1995), 267–285. [Cited on page 193]

Carol S. Dweck and Ellen L. Leggett. 1988. A social-cognitive approach to motivation and personality. *Psychological Review* 95, 2 (1988), 256–273. https://doi.org/10.1037/0033-295x.95.2.256 [Cited on pages 63 and 64]

Carol S. Dweck and David S. Yeager. 2019. Mindsets: A View From Two Eras. *Perspectives on Psychological Science* 14, 3 (Feb. 2019), 481–496. https://doi.org/10.1177/1745691618804166 [Cited on page 64]

Charles Fadel, Maya Bialik, and Bernie Trilling. 2015. *Four-Dimensional Education*. CreateSpace Independent Publishing Platform. [Cited on page 78]

Katrina Falkner and Judy Sheard. 2019. Pedagogic Approaches. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, 445–480. https://doi.org/10.1017/9781108654555.016 [Cited on pages 39, 40, and 42]

Yvon Feaster, Luke Segars, Sally K. Wahba, and Jason O. Hallstrom. 2011. Teaching CS Unplugged in the High School (with Limited Success). In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*

(Darmstadt, Germany) *(ITiCSE '11)*. ACM, New York, NY, USA, 248–252. https://doi.org/10.1145/1999747.1999817 [Cited on page 117]

Alexandre Ferreira, Eliane Pereira, Junia Anacleto, Aparecido Carvalho, and Izaura Carelli. 2008. The Common Sense-Based Educational Quiz Game Framework "What is It?". In *Proceedings of the VIII Brazilian Symposium on Human Factors in Computing Systems* (Porto Alegre, RS, Brazil) *(IHC '08)*. Sociedade Brasileira de Computação, BRA, 338–339. [Cited on page 122]

Georgios Fesakis and Kiriaki Serafeim. 2009. Influence of the Familiarization with "Scratch" on Future Teachers' Opinions and Attitudes about Programming and ICT in Education. *SIGCSE Bull.* 41, 3 (July 2009), 258–262. https://doi.org/10.1145/1595496.1562957 [Cited on page 109]

Wallace Feurzeig, Seymour Papert, Marjorie Bloom, Richard Grant, and Cynthia Solomon. 1970. Programming-Languages as a Conceptual Framework for Teaching Mathematics. *SIGCUE Outlook* 4, 2 (April 1970), 13–17. https://doi.org/10.1145/965754.965757 [Cited on pages 74 and 75]

Sally A. Fincher and Anthony V. Robins. 2019. *The Cambridge Handbook of Computing Education Research*. Cambridge University Press. https://doi.org/10.1017/9781108654555 [Cited on page 41]

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A programming environment for Scheme. *Journal of functional programming* 12, 2 (2002), 159–182. [Cited on page 108]

Abraham E. Flanigan, Markeya S. Peteranetz, Duane F. Shell, and Leen-Kiat Soh. 2015. Exploring Changes in Computer Science Students' Implicit Theories of Intelligence Across the Semester. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. Association for Computing Machinery, New York, NY, USA, 161–168. https://doi.org/10.1145/2787622.2787722 [Cited on page 67]

Robert W. Floyd. 1967. Assigning meanings to programs. *Mathematical aspects of computer science* 19 (1967), 19–32. [Cited on page 72]

Francesca Foliano, Heather Rolfe, Jonathan Buzzeo, Johnny Runge, and David Wilkinson. 2019. *Changing Mindsets: Effectiveness trial. Evaluation Report*. Technical Report. Education Endowment Foundation (EEF). https://educationendowmentfoundation.org.uk/public/files/Projects/Evaluation_Reports/Changing_Mindsets.pdf [Cited on page 65]

Jerry Lee Jr. Ford. 2009. *Scratch programming for Teens*. Course Technology. [Cited on page 109]

Luca Forlizzi, Michael Lodi, Violetta Lonati, Claudio Mirolo, Mattia Monga, Alberto Montresor, Anna Morpurgo, and Enrico Nardelli. 2018. A Core Informatics Curriculum for Italian Compulsory Education. In *Informatics in Schools. Fundamentals of Computer Science and Software Engineering. Lecture Notes in Computer Science (ISSEP 2018)*, Sergei N. Pozdniakov and Valentina Dagienė (Eds.), Vol. 11169. Springer International Publishing, Cham, 141–153. [Cited on pages x, 10, 17, 24, and 87]

George E. Forsythe. 1968. What To Do Till The Computer Scientist Comes. *The American Mathematical Monthly* 75, 5 (1968), 454–462. [Cited on page 73]

Mary L. Gick and Keith J. Holyoak. 1980. Analogical problem solving. *Cognitive Psychology* 12, 3 (July 1980), 306–355. https://doi.org/10.1016/0010-0285(80)90013-4 [Cited on page 59]

Adele Goldberg and Alan Kay. 1976. *Smalltalk-72 Instruction Manual*. Xerox. [Cited on page 106]

Catherine Good, Aneeta Rattan, and Carol S. Dweck. 2007. Adults' theories of intelligence affects feedback to males and females in math. (2007). Unpublished data, Columbia University. [Cited on page 65]

Catherine Good, Aneeta Rattan, and Carol S. Dweck. 2012. Why do women opt out? Sense of belonging and women's representation in mathematics. *Journal of Personality and Social Psychology* 102, 4 (2012), 700–717. https://doi.org/10.1037/a0026659 [Cited on page 64]

Google. [n.d.]. *Exploring Computational Thinking*. Retrieved April 4, 2017 from http://g.co/exploringct The page has now been removed, but can be found in the "CT overview" tab here: https://web.archive.org/web/20181001115843/https://edu.google.com/resources/programs/exploring-computational-thinking/. [Cited on pages 28, 30, and 31]

Stephen Gorard, Beng Huat See, and Peter Davies. 2012. *The impact of attitudes and aspirations on educational attainment and participation*. Technical Report. Joseph Rowntree Foundation. [Cited on page 65]

Saul Gorn. 1963. The computer and information sciences and the community of disciplines. *Behavioral Science* 12, 6 (1963), 433–452. [Cited on page 72]

Jamie Gorson and Eleanor O'Rourke. 2019. How Do Students Talk About Intelligence?: An Investigation of Motivation, Self-efficacy, and Mindsets in Computer Science. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. ACM, New York, NY, USA, 21–29. https://doi.org/10.1145/3291279.3339413 [Cited on pages 67 and 193]

Shuchi Grover and Roy Pea. 2013. Computational Thinking in K–12: A Review of the State of the Field. *Educational Researcher* 42, 1 (2013), 38–43. https://doi.org/10.3102/0013189X12463051 [Cited on pages 27, 33, 71, and 203]

Juan M. Gutiérrez and Ian D. Sanders. 2009. Computer Science Education in Peru: A New Kind of Monster? *SIGCSE Bull.* 41, 2 (June 2009), 86–89. https://doi.org/10.1145/1595453.1595481 [Cited on page 118]

Mark Guzdial. 2010. Does Contextualized Computing Education Help? *ACM Inroads* 1, 4 (Dec. 2010), 4–6. https://doi.org/10.1145/1869746.1869747 [Cited on page 52]

Mark Guzdial. 2015. Learner-Centered Design of Computing Education: Research on Computing for Everyone. *Synthesis Lectures on Human-Centered Informatics* 8, 6 (Nov. 2015), 1–165. https://doi.org/10.2200/s00684ed1v01y201511hci033 [Cited on pages 57, 58, 60, 61, 81, and 83]

Mark Guzdial. 2016. Brain training, like computational thinking, is unlikely to transfer to everyday problem-solving. Retrieved February 1, 2020 from https://computinged.wordpress.com/2016/03/18/brain-training-like-computational-thinking-is-unlike-to-transfer/ [Cited on page 54]

Mark Guzdial. 2019a. *A new definition of Computational Thinking: It's the Friction that we want to Minimize unless it's Generative,*. Retrieved February 1, 2020 from https://computinged.wordpress.com/2019/04/29/what-is-computational-thinking-its-the-friction-that-we-want-to-minimize/ [Cited on pages 33 and 59]

Mark Guzdial. 2019b. What's generally good for you vs what meets a need: Balancing explicit instruction vs problem/project-based learning in computer science classes. Retrieved February 1, 2020 from https://computinged.wordpress.com/2019/09/16/whats-good-for-you-vs-what-fixes-you-balancing-explicit-instruction-vs-problemproject-based-learning-in-computer-science-classes/ [Cited on pages 43 and 44]

Karin Schreier Hallett. 2016. *Building Growth Mindset Through Coding*. Retrieved February 1, 2020 from https://liquidliteracy.com/2016/12/28/building-growth-mindset-through-coding/ [Cited on page 174]

Robert K. Heinssen, Carol R. Glass, and Luanne A. Knight. 1987. Assessing computer anxiety: Development and validation of the Computer Anxiety Rating Scale. *Computers in Human Behavior* 3, 1 (Jan. 1987), 49–59. https://doi.org/10.1016/0747-5632(87)90010-0 [Cited on pages 186, 190, 193, and 228]

David Hemmendinger. 2010. A Plea for Modesty. *ACM Inroads* 1, 2 (June 2010), 4–7. https://doi.org/10.1145/1805724.1805725 [Cited on page 33]

Carl Hendrick. 2019. Schools love the idea of a growth mindset, but does it work? Retrieved February 1, 2020 from https://aeon.co/essays/schools-love-the-idea-of-a-growth-mindset-but-does-it-work [Cited on pages 64, 65, and 66]

Felienne Hermans and Efthimia Aivaloglou. 2017. To Scratch or Not to Scratch?: A Controlled Experiment Comparing Plugged First and Unplugged First Programming Lessons.

In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education* (Nijmegen, Netherlands) *(WiPSCE '17)*. ACM, New York, NY, USA, 49–56. https://doi.org/10.1145/3137065.3137072 [Cited on page 117]

Nigel Holmes. 2016. *Two mindsets*. Retrieved February 1, 2020 from http://www.nigelholmes.com/site/wp-content/uploads/2016/09/two_mindsets.png [Cited on page 177]

Juraj Hromkovič. 2006. Contributing to General Education by Teaching Informatics. In *Informatics Education – The Bridge between Using and Understanding Computers*, Roland T. Mittermeir (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 25–37. [Cited on page 18]

Juraj Hromkovič. 2016. Homo Informaticus – Why Computer Science Fundamentals are an Unavoidable Part of Human Culture and How to Teach Them. *Olympiads in Informatics* 10, 1 (July 2016), 99–109. https://doi.org/10.15388/ioi.2016.07 [Cited on page 18]

Peter Hubwieser, Michal Armoni, Torsten Brinda, Valentina Dagiene, Ira Diethelm, Michail N. Giannakos, Maria Knobelsdorf, Johannes Magenheim, Roland Mittermeir, and Sigrid Schubert. 2011. Computer Science/Informatics in Secondary Education. In *Proceedings of the 16th Annual Conference Reports on Innovation and Technology in Computer Science Education - Working Group Reports* (Darmstadt, Germany) *(ITiCSE-WGR '11)*. Association for Computing Machinery, New York, NY, USA, 19–38. https://doi.org/10.1145/2078856.2078859 [Cited on page 20]

Peter Hubwieser, Michal Armoni, Michail N. Giannakos, and Roland T. Mittermeir. 2014. Perspectives and Visions of Computer Science Education in Primary and Secondary (K-12) Schools. *ACM Trans. Comput. Educ.* 14, 2, Article 7 (June 2014), 9 pages. https://doi.org/10.1145/2602482 [Cited on page 19]

Christopher D. Hundhausen, Sean F. Farley, and Jonathan L. Brown. 2009. Can Direct Manipulation Lower the Barriers to Computer Programming and Promote Transfer of Training?: An Experimental Study. *ACM Trans. Comput. Hum. Interact.* 16, 3, Article 13 (Sept. 2009), 40 pages. https://doi.org/10.1145/1592440.1592442 [Cited on page 57]

ACM Europe & Informatics Europe. 2013. *Informatics education: Europe cannot afford to miss the boat*. Technical Report. ACM Europe & Informatics Europe. https://www.informatics-europe.org/images/documents/informatics-education-acm-ie.pdf [Cited on page 156]

ISTE and CSTA. 2011a. *Computational Thinking teacher resources*. Retrieved February 1, 2020 from https://id.iste.org/docs/ct-documents/ct-teacher-resources_2ed-pdf.pdf?sfvrsn=2 [Cited on pages 30 and 31]

ISTE and CSTA. 2011b. *Operational Definition of Computational Thinking for K-12 Education*. Retrieved February 1, 2020 from https://id.iste.org/docs/ct-documents/computational-thinking-operational-definition-flyer.pdf?sfvrsn=2 [Cited on pages 28, 29, 30, 31, and 32]

Italian Ministry of Education, University and Research. 2016. *National Plan for Digital Education*. Retrieved February 1, 2020 from http://www.istruzione.it/scuola_digitale/allegati/2016/pnsd_en.pdf [Cited on pages 23 and 35]

Italian Ministry of Education, University and Research. 2018. Indicazioni Nazionali e Nuovi Scenari. Retrieved February 1, 2020 from http://www.miur.gov.it/documents/20182/0/Indicazioni+nazionali+e+nuovi+scenari/3234ab16-1f1d-4f34-99a3-319d892a40f2 In Italian. [Cited on pages 23 and 35]

Italian Parliament. 2015. Reform of the national system of education and training (Law n.107, July 13th, 2015). https://www.gazzettaufficiale.it/eli/id/2015/07/15/15G00122/sg In Italian. [Cited on page 23]

Anita Juškevičienė and Valentina Dagienė. 2018. Computational Thinking Relationship with Digital Competence. *Informatics in Education* 17, 2 (Oct. 2018), 265–284. https://doi.org/10.15388/infedu.2018.14 [Cited on pages 28, 203, 204, and 208]

K-12 CS Framework. 2016. *K–12 Computer Science Framework*. Technical Report. http://www.k12cs.org [Cited on pages 19 and 21]

Yasmin B. Kafai and Quinn Burke. 2013. The Social Turn in K-12 Programming: Moving from Computational Thinking to Computational Participation. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 603–608. https://doi.org/10.1145/2445196.2445373 [Cited on page 19]

Yasmin B. Kafai and Quinn Burke. 2014. *Connected Code: Why Children Need to Learn Programming*. MIT Press, Cambridge, MA, USA. [Cited on page 84]

Ken Kahn. 2017. A half-century perspective on Computational Thinking. *Tecnologias, sociedade e conhecimento* 4 (Dec. 2017), 23–42. [Cited on page 104]

Antti-Juhani Kaijanaho and Ville Tirronen. 2018. Fixed Versus Growth Mindset Does Not Seem to Matter Much: A Prospective Observational Study in Two Late Bachelor Level Computer Science Courses. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/3230977.3230982 [Cited on pages 64 and 67]

Filiz Kalelioğlu. 2015. A new way of teaching programming skills to K-12 students: Code.org. *Computers in Human Behavior* 52 (Nov. 2015), 200–210. https://doi.org/10.1016/j.chb.2015.05.047 [Cited on page 60]

Filiz Kalelioğlu, Yasemin Gülbahar, and Volkan Kukul. 2016. A Framework for Computational Thinking Based on a Systematic Research Review. *Baltic Journal of Modern Computing* 4, 3 (2016), 583–596. [Cited on page 207]

Petra Kastl, Ulrich Kiesmüller, and Ralf Romeike. 2016. Starting out with Projects: Experiences with Agile Software Development in High Schools. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (Münster, Germany) *(WiPSCE '16)*. Association for Computing Machinery, New York, NY, USA, 60–65. https://doi.org/10.1145/2978249.2978257 [Cited on page 121]

Donald L. Katz. 1960. Conference Report on the Use of Computers in Engineering Classroom Instruction. *Commun. ACM* 3, 10 (Oct. 1960), 522–527. https://doi.org/10.1145/367415.993453 [Cited on page 72]

Alan Kay, Kim Rose, Dan Ingalls, Ted Kaehler, John Maloney, and Scott Wallace. 1997. *Etoys & SimStories*. Technical Report. ImagiLearning Group, Walt Disney Imagineering. http://www.vpri.org/pdf/hc_etoys_sim_1997.pdf [Cited on page 107]

Alan C. Kay. 1993. The Early History of Smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA) *(HOPL-II)*. Association for Computing Machinery, New York, NY, USA, 69–95. https://doi.org/10.1145/154766.155364 [Cited on page 106]

Elizabeth R. Kazakoff, Amanda Sullivan, and Marina U. Bers. 2012. The Effect of a Classroom-Based Intensive Robotics and Programming Workshop on Sequencing Ability in Early Childhood. *Early Childhood Education Journal* 41, 4 (Oct. 2012), 245–255. https://doi.org/10.1007/s10643-012-0554-5 [Cited on page 60]

Khan Academy and PERTS. 2014. *Growth Mindset Lesson Plan*. Retrieved February 1, 2020 from https://cdn.kastatic.org/KA-share/Toolkit-photos/FINAL+Growth+Mindset+Lesson+Plan.pdf [Cited on pages 177 and 226]

Paul A. Kirschner, John Sweller, and Richard E. Clark. 2006. Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching. *Educational Psychologist* 41, 2 (June 2006), 75–86. https://doi.org/10.1207/s15326985ep4102_1 [Cited on pages 42 and 57]

David Klahr and Sharon McCoy Carver. 1988. Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology* 20, 3 (1988), 362–404. https://doi.org/10.1016/0010-0285(88)90004-7 [Cited on page 58]

Donald E. Knuth. 1972. George Forsythe and the Development of Computer Science. *Commun. ACM* 15, 8 (Aug. 1972), 721–726. https://doi.org/10.1145/361532.361538 [Cited on page 73]

Donald E. Knuth. 1974. Computer Science and Its Relation to Mathematics. *The American Mathematical Monthly* 81, 4 (1974), 323–343. [Cited on page 73]

Donald E. Knuth. 1997. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd Edition*. Addison-Wesley Professional. [Cited on pages 101 and 102]

Alfie Kohn. 2015. *The perils of "Growth Mindset" education: Why we're trying to fix our kids when we should be fixing the system*. Retrieved February 1, 2020 from https://www.salon.com/2015/08/16/the_education_fad_thats_hurting_our_kids_what_you_need_to_know_about_growth_mindset_theory_and_the_harmful_lessons_it_imparts/ [Cited on page 66]

Michael Kölling, Neil Brown, and Amjad Altadmri. 2017. Frame-Based Editing. *Journal of Visual Languages and Sentient Systems* 3, 1 (July 2017), 40–67. https://doi.org/10.18293/vlss2017-009 [Cited on pages 113 and 114]

Jane Krauss and Kiki Prottsman. 2016. *Computational Thinking and Coding for Every Student: The Teacher's Getting-Started Guide*. Corwin Press. [Cited on page 208]

Thomas E. Kurtz. 1978. BASIC. *SIGPLAN Not.* 13, 8 (Aug. 1978), 103–118. https://doi.org/10.1145/960118.808376 [Cited on page 107]

Bruno Latour. 1986. Visualisation and Cognition: Thinking with Eyes and Hands. In *Knowledge and Society Studies in the Sociology of Culture Past and Present*, H. Kuklick (Ed.). Vol. 6. Jai Press, 1–40. [Cited on page 72]

David C. Leonard. 2002. *Learning Theories: A to Z*. Greenwood - ABC-CLIO. [Cited on page 37]

Sarah-Jane Leslie, Andrei Cimpian, Meredith Meyer, and Edward Freeland. 2015. Expectations of brilliance underlie gender distributions across academic disciplines. *Science* 347, 6219 (2015), 262–265. https://doi.org/10.1126/science.1261375 [Cited on page 193]

Olivia Levrini, Eleonora Barelli, Michael Lodi, Giovanni Ravaioli, Giulia Tasquier, Laura Branchetti, Michela Clementi, Paola Fantini, and Fabio Filippi. 2018. The perspective of complexity to futurize STEM education: an interdisciplinary module on Artificial Intelligence. In *Abstracts of GIREP-MPTL conference* (Donostia-San Sebastian, Spain, 2018-07). 2 pages. https://www.girep2018.com/contenidos/files/abstracts/resumen/autor/178_abs_con_v1.pdf [Cited on page xi]

Arthur Lewis and David Smith. 1993. Defining higher order thinking. *Theory Into Practice* 32, 3 (June 1993), 131–137. https://doi.org/10.1080/00405849309543588 [Cited on page 53]

Clayton Lewis. 2007. Attitudes and Beliefs about Computer Science among Students and Faculty. *SIGCSE Bull.* 39, 2 (June 2007), 37–41. https://doi.org/10.1145/1272848.1272880 [Cited on page 66]

Colleen M. Lewis. 2010. How Programming Environment Shapes Perception, Learning and Goals: Logo vs. Scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10)*. ACM, New York, NY, USA, 346–350. https://doi.org/10.1145/1734263.1734383 [Cited on page 112]

Colleen M. Lewis. 2017. Good (and Bad) Reasons to Teach All Students Computer Science. In *New Directions for Computing Education: Embedding Computing Across Disciplines*,

Samuel B. Fee, Amanda M. Holland-Minkley, and Thomas E. Lombardi (Eds.). Springer International Publishing, Cham, 15–34. https://doi.org/10.1007/978-3-319-54226-3_2 [Cited on pages 33, 61, 68, 83, 84, 174, and 183]

Colleen M. Lewis, Ruth E. Anderson, and Ken Yasuhara. 2016. "I Don't Code All Day": Fitting in Computer Science When the Stereotypes Don't Fit. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/2960310.2960332 [Cited on pages 4 and 63]

Colleen M. Lewis, Sarah Esper, Victor Bhattacharyya, Noelle Fa-Kaji, Neftali Dominguez, and Arielle Schlesinger. 2014. Children's Perceptions of What Counts as a Programming Language. *J. Comput. Sci. Coll.* 29, 4 (April 2014), 123–133. [Cited on page 112]

Colleen M. Lewis, Ken Yasuhara, and Ruth E. Anderson. 2011. Deciding to Major in Computer Science: A Grounded Theory of Students' Self-assessment of Ability. In *Proceedings of the Seventh International Workshop on Computing Education Research* (Providence, Rhode Island, USA) *(ICER '11)*. ACM, New York, NY, USA, 3–10. https://doi.org/10.1145/2016911.2016915 [Cited on page 67]

Yuen-Kuang Cliff Liao and George W. Bright. 1991. Effects of Computer Programming on Cognitive Outcomes: A Meta-Analysis. *Journal of Educational Computing Research* 7, 3 (Aug. 1991), 251–268. https://doi.org/10.2190/e53g-hh8k-ajrr-k69m [Cited on page 58]

Michael Lodi. 2014a. *Imparare il pensiero computazionale, imparare a programmare*. Master's thesis. Alma Mater Studiorum - Università di Bologna. http://amslaurea.unibo.it/6730/ In Italian. [Cited on page ix]

Michael Lodi. 2014b. Learn computational thinking, learn to program. *Mondo Digitale* 13, 51 (2014), 822–833. http://mondodigitale.aicanet.net/2014-3/03_Computational_Thinking/03_23.pdf In Italian. [Cited on page ix]

Michael Lodi. 2016. Pensare come un informatico non vuol dire pensare come un computer. In *Coding pensiero computazionale nella scuola primaria*. Marco Giordano and Caterina Moscetti (Authors). ELI La Spiga. Preface. In Italian. [Cited on page xii]

Michael Lodi. 2017. Growth Mindset in Computational Thinking Teaching and Teacher Training. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. ACM, New York, NY, USA, 281–282. https://doi.org/10.1145/3105726.3105736 [Cited on pages xi and 12]

Michael Lodi. 2018a. Can Creative Computing Foster Growth Mindset?. In *Joint Proceedings of the 1st Co-Creation in the Design, Development and Implementation of Technology-Enhanced Learning workshop (CC-TEL 2018) and Systems of Assessments for Computational Thinking Learning workshop (TACKLE 2018) co-located with 13th European Conference on Technology Enhanced Learning (ECTEL 2018), Leeds, United Kingdom, September 3rd, 2018. (CEUR Workshop Proceedings)*, Alicja Piotrkowicz,

Rosie Dent-Spargo, Sebastian Dennerlein, István Koren, Panagiotis Antoniou, Paul Bailey, Tamsin Treasure-Jones, Ilenia Fronza, and Claus Pahl (Eds.), Vol. 2190. CEUR-WS.org. http://ceur-ws.org/Vol-2190/TACKLE_2018_paper_3.pdf [Cited on pages x, 12, 185, and 227]

Michael Lodi. 2018b. Pensiero Computazionale: dalle "scuole di samba della computazione" ai CoderDojo. *Mondo Digitale* 17, 77 (2018). http://mondodigitale.aicanet.net/2018-4/02_Pensiero_Computazionale,_Coding,_Making_e_Robotica_Educativa/04_Pensiero_Computazionale.pdf Abstract. In Italian. [Cited on page xi]

Michael Lodi. 2018c. Pensiero Computazionale: dalle "scuole di samba della computazione" ai CoderDojo. In *Atti del convegno DIDAMATICA 2018*. AICA, Cesena, Italy. https://www.aicanet.it/documents/10776/2101882/didamatica2018_paper_57.pdf In Italian. [Cited on pages x, xi, 9, and 46]

Michael Lodi. 2019. Does Studying CS Automatically Foster a Growth Mindset?. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland, UK) *(ITiCSE '19)*. Association for Computing Machinery, New York, NY, USA, 147–153. https://doi.org/10.1145/3304221.3319750 [Cited on pages x, 12, 17, 173, and 223]

Michael Lodi, Renzo Davoli, Rebecca Montanari, and Simone Martini. 2020. Informatica senza e con computer nella Scuola Primaria. In *Coding e oltre: informatica nella scuola*, Enrico Nardelli (Ed.). Lisciani. In Italian. To appear. [Cited on pages xi, 10, and 125]

Michael Lodi, Dario Malchiodi, Mattia Monga, Anna Morpurgo, and Bernadette Spieler. 2019. Constructionist Attempts at Supporting the Learning of Computer Programming: A Survey. *Olympiads in Informatics* 13 (July 2019), 99–121. https://doi.org/10.15388/ioi.2019.07 [Cited on pages ix, 10, and 97]

Michael Lodi and Simone Martini. [n.d.]. Computational Thinking, between Papert and Wing. ([n. d.]). Unpublished. [Cited on pages xii and 9]

Michael Lodi, Simone Martini, and Enrico Nardelli. 2017. Do we really need computational thinking? *Mondo Digitale* 72, Article 2 (Nov. 2017), 15 pages. http://mondodigitale.aicanet.net/2017-5/articoli/MD72_02_abbiamo_davvero_bisogno_del_pensiero_computazionale.pdf In Italian. [Cited on pages ix, 9, 17, 80, and 170]

Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. ACM, New York, NY, USA, 1449–1461. https://doi.org/10.1145/2858036.2858252 [Cited on page 68]

Jay Lynch. 2018. *Growth Mindset: The Perils of a Good Research Story*. Retrieved February 1, 2020 from https://medium.com/@quixotic_scholar/growth-mindset-the-perils-of-a-good-research-story-d6ce32a447d2 [Cited on page 65]

John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 16 (Nov. 2010), 15 pages. https://doi.org/10.1145/1868358.1868363 [Cited on pages 109 and 110]

Linda Mannila, Valentina Dagienė, Barbara Demo, Natasa Grgurina, Claudio Mirolo, Lennart Rolandsson, and Amber Settle. 2014. Computational Thinking in K-9 Education. In *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference* (Uppsala, Sweden) *(ITiCSE-WGR '14)*. Association for Computing Machinery, New York, NY, USA, 1–29. https://doi.org/10.1145/2713609.2713610 [Cited on page 82]

Rita Marchignoli and Michael Lodi. 2016. *EAS e pensiero computazionale*. ELS LA SCUOLA. In Italian. [Cited on page xi]

Lauren E. Margulieux, Brian Dorn, and Kristin A. Searle. 2019. Learning Sciences for Computing Education. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, 208–230. https://doi.org/10.1017/9781108654555.009 [Cited on page 42]

Maria A. Martinez, Narcis Sauleda, and Güenter L. Huber. 2001. Metaphors as blueprints of thinking about teaching and learning. *Teaching and Teacher Education* 17, 8 (Nov. 2001), 965–977. https://doi.org/10.1016/s0742-051x(01)00043-9 [Cited on page 39]

Simone Martini. 2012. Lingua Universalis. *Annali della Pubblica Istruzione* 4-5 (2012), 65–70. In Italian. [Cited on page 33]

Joseph J. Martocchio. 1994. Effects of conceptions of ability on anxiety, self-efficacy, and learning in training. *Journal of Applied Psychology* 79, 6 (1994), 819–825. https://doi.org/10.1037/0021-9010.79.6.819 [Cited on page 186]

Michael R. Matthews. 1997. Introductory Comments on Philosophy and Constructivism in Science Education. *Science & Education* 6, 1 (01 Jan. 1997), 5–14. https://doi.org/10.1023/A:1008650823980 [Cited on page 42]

Matthew M. Maurer. 1994. Computer anxiety correlates and what they tell us: A literature review. *Computers in Human Behavior* 10, 3 (Sept. 1994), 369–376. https://doi.org/10.1016/0747-5632(94)90062-0 [Cited on page 186]

Richard E. Mayer, Jennifer L. Dyck, and William Vilberg. 1986. Learning to Program and Learning to Think: What's the Connection? *Commun. ACM* 29, 7 (July 1986), 605–610. https://doi.org/10.1145/6138.6142 [Cited on page 74]

Jacques Mazoyer. 2005. Universalité de la notion de calcul. L'enseignement de l'informatique de la maternelle à la terminale, Académie des sciences – workshop. [Cited on page 213]

John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199 [Cited on page 72]

John McCarthy. 1961. A Basis for a Mathematical Theory of Computation, Preliminary Report. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference* (Los Angeles, California) *(IRE-AIEE-ACM '61 (Western))*. Association for Computing Machinery, New York, NY, USA, 225–238. https://doi.org/10.1145/1460690.1460715 [Cited on page 72]

Robert McCartney and Josh Tenenberg (Eds.). 2014. Special Issue on Computing Education in K-12 Schools. *ACM Trans. Comput. Educ.* 14, 2 (2014). [Cited on page 20]

Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. 2013. Learning computer science concepts with Scratch. *Computer Science Education* 23, 3 (2013), 239–264. https://doi.org/10.1080/08993408.2013.832022 [Cited on page 104]

Guy Merchant. 2007. Mind the Gap(s): Discourses and Discontinuity in Digital Literacies. *E-Learning and Digital Media* 4, 3 (2007), 241–255. [Cited on page 79]

Marvin Minsky. 1970. Form and Content in Computer Science (1970 ACM Turing Lecture). *J. ACM* 17, 2 (April 1970), 197–215. https://doi.org/10.1145/321574.321575 [Cited on pages 73 and 74]

Marcello Missiroli, Daniel Russo, and Paolo Ciancarini. 2016. Learning Agile Software Development in High School: An Investigation. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 293–302. https://doi.org/10.1145/2889160.2889180 [Cited on page 121]

Mattia Monga, Michael Lodi, Dario Malchiodi, Anna Morpurgo, and Bernadette Spieler. 2018. Learning to program in a constructionist way. In *Proceedings of Constructionism 2018: Constructionism, Computational thinking and Educational Innovation* (Vilnius, Lithuania). 901–924. http://www.constructionism2018.fsf.vu.lt/file/repository/Proceeding_2018_Constructionism.pdf [Cited on pages x, 10, and 97]

Patricia Morreale and David Joiner. 2011. Reaching Future Computer Scientists. *Commun. ACM* 54, 4 (April 2011), 121–124. https://doi.org/10.1145/1924421.1924448 [Cited on page 118]

Claudia M. Mueller and Carol S. Dweck. 1998. Praise for intelligence can undermine children's motivation and performance. *Journal of Personality and Social Psychology* 75, 1 (1998), 33–52. https://doi.org/10.1037/0022-3514.75.1.33 [Cited on page 64]

Laurie Murphy and Lynda Thomas. 2008. Dangers of a Fixed Mindset: Implications of Self-Theories Research for Computer Science Education. *SIGCSE Bull.* 40, 3 (June 2008), 271–275. https://doi.org/10.1145/1597849.1384344 [Cited on pages 66 and 183]

Enrico Nardelli. 2019. Do we really need computational thinking? *Commun. ACM* 62, 2 (Jan. 2019), 32–35. https://doi.org/10.1145/3231587 [Cited on pages 80 and 170]

Enrico Nardelli, Luca Forlizzi, Michael Lodi, Violetta Lonati, Claudio Mirolo, Mattia Monga, Alberto Montresor, and Anna Morpurgo. 2017. *Proposal for a national Informatics curriculum in the Italian school*. Technical Report. CINI. https://www.consorzio-cini.it/images/PROPOSAL-Informatics-curriculum-Italian-school.pdf [Cited on pages x, 10, 24, 87, and 211]

National Research Council. 2000. *How People Learn: Brain, Mind, Experience, and School: Expanded Edition*. The National Academies Press, Washington, DC. https://doi.org/10.17226/9853 [Cited on pages 51, 53, and 55]

Peter Naur. 1966. Proof of algorithms by general snapshots. *BIT Numerical Mathematics* 6, 4 (1966), 310–316. [Cited on page 72]

Allen Newell, Alan J. Perlis, and Herbert A. Simon. 1967. Computer Science. *Science* 157, 3795 (1967), 1373–1374. [Cited on page 73]

Allen Newell and Herbert A. Simon. 1972. *Human Problem Solving*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. [Cited on page 53]

David Nofre, Mark Priestley, and Gerard Alberts. 2014. When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950–1960. *Technology and Culture* 55, 1 (2014), 40–75. http://www.jstor.org/stable/24468397 [Cited on page 72]

Jum C. Nunnally. 1978. *Psychometric Theory: 2d Ed*. McGraw-Hill. [Cited on pages 178 and 192]

David B. Palumbo. 1990. Programming Language/Problem-Solving Research: A Review of Relevant Issues. *Review of Educational Research* 60, 1 (March 1990), 65–89. https://doi.org/10.3102/00346543060001065 [Cited on page 58]

David B. Palumbo and W. Michael Reed. 1991. The Effect of BASIC Programming Language Instruction on High School Students' Problem Solving Ability and Computer Anxiety. *Journal of Research on Computing in Education* 23, 3 (March 1991), 343–372. https://doi.org/10.1080/08886504.1991.10781967 [Cited on page 58]

Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA. [Cited on pages 4, 27, 44, 45, 74, 75, 76, 104, 106, and 119]

Seymour Papert. 1987. Information Technology and Education: Computer Criticism vs. Technocentric Thinking. *Educational Researcher* 16, 1 (Jan. 1987), 22–30. https://doi.org/10.3102/0013189x016001022 [Cited on page 58]

Seymour Papert. 1993. *The Children's Machine: Rethinking School in the Age of the Computer*. Basic Books, Inc., New York, NY, USA. [Cited on pages 47 and 77]

Seymour Papert. 1996a. *The connected family: Bridging the digital generation gap*. Long Street Press, Atlanta, GA. [Cited on page 83]

Seymour Papert. 1996b. An Exploration in the Space of Mathematics Educations. *International Journal of Computers for Mathematical Learning* 1, 1 (1996), 95–123. [Cited on page 77]

Seymour Papert. 2000. What's the Big Idea? Toward a Pedagogy of Idea Power. *IBM Systems Journal* 39, 3-4 (July 2000), 720–729. https://doi.org/10.1147/sj.393.0720 [Cited on pages 75, 76, and 77]

Seymour Papert. 2006. Keynote lecture. http://dailypapert.com/wp-content/uploads/2012/05/Seymour-Vietnam-Talk-2006.pdf Keynote at ICMI 17 Conference in Hanoi, Viet Nam. [Cited on pages 78 and 83]

Seymour Papert. 80s. *Constructionism vs. Insructionism*. Retrieved February 1, 2020 from http://papert.org/articles/const_inst/const_inst1.html Speech delivered in 1980s to a conference of educators in Japan. [Cited on page 45]

Seymour Papert and Idit Harel. 1991. Situating Constructionism. In *Constructionism*, Seymour Papert and Idit Harel (Eds.). Ablex Publishing Corporation, Norwood, NJ, Chapter 1. [Cited on pages 44, 45, and 77]

Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. 2016. Evidence That Computer Science Grades Are Not Bimodal. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 113–121. https://doi.org/10.1145/2960310.2960312 [Cited on pages 4, 63, and 183]

Richard E. Pattis. 1981. *Karel The Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons. [Cited on page 108]

Roy D. Pea and D.Midian Kurland. 1984. On the cognitive effects of learning computer programming. *New Ideas in Psychology* 2, 2 (Jan. 1984), 137–168. https://doi.org/10.1016/0732-118x(84)90018-7 [Cited on pages 58 and 74]

Jean Piaget. 1973. *To Understand is to Invent: The Future of Education*. Penguin Books. [Cited on page 44]

Plato. 1982. *The Dialogues of Plato translated into English with Analyses and Introductions by B. Jowett, M.A. in Five Volumes. 3rd edition revised and corrected*. Oxford University Press. https://oll.libertyfund.org/titles/166 [Cited on page 53]

Kiki Prottsman. 2015. *Coding vs. Programming - Battle of the Terms!* Retrieved February 1, 2020 from http://www.huffingtonpost.com/kiki-prottsman/coding-vs-programming-bat_b_7042816.html [Cited on page 34]

Kevin J. Pugh and David A. Bergin. 2006. Motivational Influences on Transfer. *Educational Psychologist* 41, 3 (Sept. 2006), 147–160. https://doi.org/10.1207/s15326985ep4103_2 [Cited on page 61]

George Pólya. 1945. *How to Solve It*. Princeton University Press. [Cited on page 53]

Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education* 18, 1, Article 1 (Oct. 2017), 24 pages. https://doi.org/10.1145/3077618 [Cited on page 102]

R Core Team. 2013. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/ [Cited on pages 178 and 190]

Giovanni Ravaioli, Eleonora Barelli, Laura Branchetti, Michael Lodi, Sara Satanassi, and Olivia Levrini. 2019. Epistemological Activators To Value S-T-E-M Concepts For Education. In *Proceedings the 13th Biennial Conference of the European Science Education Research Association* (Bologna, Italy) *(ESERA '19)*. Oral presentation. [Cited on page xi]

Kenneth J. Reid and Daniel M. Ferguson. 2014. Do design experiences in engineering build a "growth mindset" in students?. In *2014 IEEE Integrated STEM Education Conference*. 1–5. https://doi.org/10.1109/ISECon.2014.6891046 [Cited on pages 68 and 183]

Mitchel Resnick. 1996. Distributed Constructionism. In *Proceedings of the 1996 International Conference on Learning Sciences* (Evanston, Illinois) *(ICLS '96)*. International Society of the Learning Sciences, 280–284. [Cited on page 122]

Mitchel Resnick. 2007. All I Really Need to Know (about Creative Thinking) I Learned (by Studying How Children Learn) in Kindergarten. In *Proceedings of the 6th ACM SIGCHI Conference on Creativity & Cognition* (Washington, DC, USA) *(C&C '07)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/1254960.1254961 [Cited on pages 46, 48, and 122]

Mitchel Resnick. 2014. Give P's a chance: Projects, Peers, Passion, Play. In *Proceedings of Constructionism and Creativity Conference*. Vienna, Austria, 13–20. [Cited on page 48]

Mitchel Resnick. 2017a. *Lifelong Kindergarten: Cultivating Creativity Through Projects, Passion, Peers, and Play*. MIT Press, Cambridge, MA, USA. [Cited on pages 46, 48, and 93]

Mitchel Resnick. 2017b. The Seeds That Seymour Sowed. *International Journal of Child-Computer* (2017). http://web.media.mit.edu/~mres/papers/IJCCI-seeds-seymour-sowed.pdf [Cited on page 48]

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779 [Cited on pages 19 and 111]

Mitchel Resnick and David Siegel. 2015. A Different Approach to Coding. *Bright/Medium* (2015). https://brightthemag.com/a-different-approach-to-coding-d679b06d83a [Cited on page 144]

John W. Rice. 2012. The Gamification of Learning and Instruction: Game-Based Methods and Strategies for Training and Education. *Int. J. Gaming Comput. Mediat. Simul.* 4, 4 (Oct. 2012), 81–83. https://doi.org/10.4018/jgcms.2012100106 [Cited on page 39]

Anthony V. Robins. 2019. Novice Programmers and Introductory Programming. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, 327–376. https://doi.org/10.1017/9781108654555.013 [Cited on page 63]

Anthony V. Robins, Lauren E. Margulieux, and Briana B. Morrison. 2019. Cognitive Sciences for Computing Education. In *The Cambridge Handbook of Computing Education Research*. Cambridge University Press, 231–275. https://doi.org/10.1017/9781108654555.010 [Cited on pages 39, 51, 52, and 57]

Susan H. Rodger, Jenna Hayes, Gaetjens Lezin, Henry Qin, Deborah Nelson, Ruth Tucker, Mercedes Lopez, Stephen Cooper, Wanda Dann, and Don Slater. 2009. Engaging Middle School Teachers and Students with Alice in a Diverse Set of Subjects. *SIGCSE Bull.* 41, 1 (March 2009), 271–275. https://doi.org/10.1145/1539024.1508967 [Cited on page 111]

Giovanni Sala and Fernand Gobet. 2017. Does Far Transfer Exist? Negative Evidence From Chess, Music, and Working Memory Training. *Current Directions in Psychological Science* 26, 6 (2017), 515–520. https://doi.org/10.1177/0963721417712760 [Cited on page 54]

Katie Salen and Eric Zimmerman. 2003. *Rules of Play: Game Design Fundamentals*. The MIT Press. [Cited on page 122]

Gavriel Salomon. 1984. On ability development and far transfer: A response to Pea and Kurland. *New Ideas in Psychology* 2, 2 (1984), 169–174. [Cited on page 74]

Hong Kian Sam, Abang Ekhsan Abang Othman, and Zaimuarifuddin Shukri Nordin. 2005. Computer Self-Efficacy, Computer Anxiety, and Attitudes toward the Internet: A Study among Undergraduates in Unimas. *Journal of Educational Technology & Society* 8, 4 (2005), 205–219. http://www.jstor.org/stable/jeductechsoci.8.4.205 [Cited on pages 186, 190, and 228]

Jaime Sánchez and Ruby Olivares. 2011. Problem solving and collaboration using mobile serious games. *Computers & Education* 57, 3 (Nov. 2011), 1943–1952. https://doi.org/10.1016/j.compedu.2011.04.012 [Cited on page 122]

Emmanuel Schanzer, Kathi Fisler, and Shriram Krishnamurthi. 2018. Assessing Bootstrap: Algebra Students on Scaffolded and Unscaffolded Word Problems. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 8–13. https://doi.org/10.1145/3159450.3159498 [Cited on pages 60 and 109]

Ronny Scherer. 2016. Learning from the Past - The Need for Empirical Evidence on the Transfer Effects of Computer Programming Skills. *Frontiers in Psychology* 7 (Sept. 2016). https://doi.org/10.3389/fpsyg.2016.01390 [Cited on pages 59 and 83]

Ronny Scherer, Fazilat Siddiq, and Bárbara Sánchez Viveros. 2019. The cognitive benefits of learning computer programming: A meta-analysis of transfer effects. *Journal of Educational Psychology* 111, 5 (July 2019), 764–792. https://psycnet.apa.org/doi/10.1037/edu0000314 [Cited on page 61]

Alan H. Schoenfeld. 1985. *Mathematical Problem Solving*. Elsevier. https://doi.org/10.1016/c2013-0-05012-8 [Cited on page 54]

Carsten Schulte. 2013. Reflections on the Role of Programming in Primary and Secondary Computing Education. In *Proceedings of the 8th Workshop in Primary and Secondary Computing Education* (Aarhus, Denmark) *(WiPSE '13)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/2532748.2532754 [Cited on page 19]

Daniel L. Schwartz and John D. Bransford. 1998. A Time For Telling. *Cognition and Instruction* 16, 4 (Dec. 1998), 475–5223. https://doi.org/10.1207/s1532690xci1604_4 [Cited on page 57]

Michael J. Scott and Gheorghita Ghinea. 2014. On the Domain-Specificity of Mindsets: The Relationship Between Aptitude Beliefs and Programming Practice. *IEEE Transactions on Education* 57, 3 (Aug. 2014), 169–174. https://doi.org/10.1109/TE.2013.2288700 [Cited on page 67]

ScratchEd Team. 2013. *Creative Computing Online Workshop*. Retrieved February 1, 2020 from https://creative-computing.appspot.com/preview [Cited on page 111]

Deborah Seehorn, Stephen Carey, Brian Fuschetto, Irene Lee, Daniel Moix, Dianne O'Grady-Cunniff, Barbara Boucher Owens, Chris Stephenson, and Anita Verno. 2011. *CSTA K–12 Computer Science Standards: Revised 2011*. Technical Report. Association for Computing Machinery, New York, NY, USA. [Cited on page 88]

Cynthia Selby and John Woollard. 2013. *Computational thinking: the developing definition*. Project Report. University of Southampton. https://eprints.soton.ac.uk/356481/ [Cited on page 203]

Sue Sentance. 2018. Introducing Why Teach Computer Science in School. In *Computer Science Education. Perspectives on Teaching and Learning in School*, S. Sentance, E. Barendsen, and C. Schulte (Eds.). Bloomsbury Academic, London, Chapter 1, 3–4. [Cited on page 34]

Sue Sentance and Andrew Csizmadia. 2017. Computing in the curriculum: Challenges and strategies from a teacher's perspective. *Education and Information Technologies* 22, 2 (01 March 2017), 469–495. https://doi.org/10.1007/s10639-016-9482-0 [Cited on page 118]

Sue Sentance and Jane Waite. 2018. Computing in the classroom: Tales from the chalkface. *it - Information Technology* 60, 2 (April 2018), 103–112. https://doi.org/10.1515/itit-2017-0014 [Cited on page 60]

Sentis. 2012. *Neuroplasticity*. Retrieved February 1, 2020 from https://www.youtube.com/watch?v=ELpfYCZa87g [Cited on page 177]

Russell Shackelford, Andrew McGettrick, Robert Sloan, Heikki Topi, Gordon Davies, Reza Kamali, James Cross, John Impagliazzo, Richard LeBlanc, and Barry Lunt. 2006. Computing Curricula 2005: The Overview Report. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (Houston, Texas, USA) *(SIGCSE '06)*. Association for Computing Machinery, New York, NY, USA, 456–457. https://doi.org/10.1145/1121341.1121482 [Cited on page 2]

Lee S. Shulman. 1986. Those who understand: Knowledge growth in teaching. *Educational Researcher* 15, 2 (1986), 4–14. [Cited on page 94]

Lee S. Shulman. 1987. Knowledge and Teaching: Foundations of the New Reform. *Harvard Educational Review* 57, 1 (1987), 1–23. https://doi.org/10.17763/haer.57.1.j463w79r56455411 [Cited on page 3]

Valerie J. Shute, Chen Sun, and Jodi Asbell-Clarke. 2017. Demystifying computational thinking. *Educational Research Review* 22 (2017), 142–158. https://doi.org/10.1016/j.edurev.2017.09.003 [Cited on page 206]

Beth Simon, Brian Hanks, Laurie Murphy, Sue Fitzgerald, Renée McCauley, Lynda Thomas, and Carol Zander. 2008. Saying Isn't Necessarily Believing: Influencing Self-Theories in Computing. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 173–184. https://doi.org/10.1145/1404520.1404537 [Cited on pages 67, 177, 183, and 226]

Teemu Sirkiä. 2012. *Recognizing Programming Misconceptions: An Analysis of the Data Collected from the UUhistle Program Simulation Tool*. Master's thesis. Department of Computer Science and Engineering, Aalto University. [Cited on page 102]

Victoria F. Sisk, Alexander P. Burgoyne, Jingze Sun, Jennifer L. Butler, and Brooke N. Macnamara. 2018. To What Extent and Under Which Circumstances Are Growth Mind-Sets Important to Academic Achievement? Two Meta-Analyses. *Psychological Science* 29, 4 (2018), 549–571. https://doi.org/10.1177/0956797617739704 [Cited on page 65]

Kelly Smith. 2016. *Why code club is the best way to develop a growth mindset*. Retrieved February 1, 2020 from https://medium.com/coding-at-the-library/why-code-club-is-the-best-way-to-develop-a-growth-mindset-2ac354ab7322 [Cited on page 174]

Neil Smith, Yasemin Allsop, Helen Caldwell, David Hill, Yota Dimitriadi, and Andrew Paul Csizmadia. 2015. Master Teachers in Computing: What Have We Achieved?. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (London, United Kingdom) *(WiPSCE '15)*. Association for Computing Machinery, New York, NY, USA, 21–24. https://doi.org/10.1145/2818314.2818332 [Cited on page 118]

Juha Sorva. 2012. *Visual program simulation in introductory programming education*. Ph.D. Dissertation. Aalto University. [Cited on pages 40 and 41]

Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2, Article 8 (July 2013), 31 pages. https://doi.org/10.1145/2483710.2483713 [Cited on pages 100, 101, 102, and 146]

George A. Stanic. 1986. Mental Discipline Theory and Mathematics Education. *For the Learning of Mathematics* 6, 1 (1986), 39–47. http://www.jstor.org/stable/40247802 [Cited on page 53]

Katherine L. Sun. 2015. *There's no limit: mathematics teaching for a growth mindset*. Ph.D. Dissertation. Graduate School of Education, Stanford University. http://purl.stanford.edu/xf479cc2194 [Cited on pages 65, 176, 178, and 224]

Keith S. Taber. 2012. Constructivism as educational theory: Contingency in learning, and optimally guided instruction. In *Educational theory*, Hassaskhah Jaleh (Ed.). Nova, New York, 39–61. [Cited on page 43]

Rivka Taub, Michal Armoni, and Mordechai Ben-Ari. 2012. CS Unplugged and Middle-School Students' Views, Attitudes, and Intentions Regarding CS. *ACM Trans. Comput. Educ.* 12, 2, Article 8 (April 2012), 29 pages. https://doi.org/10.1145/2160547.2160551 [Cited on page 117]

Matti Tedre. 2014. *The Science of Computing: Shaping a Discipline*. Chapman and Hall/CRC. [Cited on pages 72 and 81]

Matti Tedre and Peter J. Denning. 2016. The Long Quest for Computational Thinking. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '16)*. Association for Computing Machinery, New York, NY, USA, 120–129. https://doi.org/10.1145/2999541.2999542 [Cited on pages 71, 72, 75, and 79]

The Committee on European Computing Education (CECE). 2017. *Informatics Education in Europe: Are We all in the Same Boat?* Technical Report. ACM Europe & Informatics Europe. http://www.informatics-europe.org/component/phocadownload/category/10-reports.html?download=60:cece-report [Cited on pages 94 and 170]

The Royal Society. 2012. *Shut down or restart? The way forward for computing in UK schools*. Technical Report. The Royal Society, London. https://royalsociety.org/~/media/education/computing-in-schools/2012-01-12-computing-in-schools.pdf [Cited on pages 19 and 88]

The Royal Society. 2017. *After the reboot: Computing education in UK schools*. Technical Report. The Royal Society, London. https://royalsociety.org/~/media/policy/projects/computing-education/computing-education-report.pdf [Cited on pages 2, 94, and 170]

Renate Thies and Jan Vahrenhold. 2013. On Plugging "Unplugged" into CS Classes. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*

(Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 365–370. https://doi.org/10.1145/2445196.2445303 [Cited on page 117]

Edward L. Thorndike. 1924. Mental Discipline in High School Studies. *Journal of Educational Psychology* 15, 2 (1924), 83–98. https://doi.org/10.1037/h0071035 [Cited on page 53]

Edward L. Thorndike and Robert S. Woodworth. 1901. The influence of improvement in one mental function upon the efficiency of other functions (I). *Psychological Review* 8, 3 (May 1901), 247–261. https://doi.org/10.1037/h0074898 [Cited on pages 53 and 54]

Sigmund Tobias and Thomas M. Duffy (Eds.). 2009. *Constructivist instruction: Success or failure?* Routledge. [Cited on pages 43 and 57]

André Tricot and John Sweller. 2013. Domain-Specific Knowledge and Why Teaching Generic Skills Does Not Work. *Educational Psychology Review* 26, 2 (Oct. 2013), 265–283. https://doi.org/10.1007/s10648-013-9243-1 [Cited on pages 54 and 55]

Nath Tumlin. 2017. Teacher Configurable Coding Challenges for Block Languages. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 783–784. https://doi.org/10.1145/3017680.3022467 [Cited on page 104]

Annette Vee. 2013. Understanding Computer Programming as a Literacy. *Literacy in Composition Studies* 1, 2 (2013), 42–64. [Cited on page 19]

Arjun Venkataswamy. 2016. *How Learning to Code Can Help Develop a Growth Mindset*. Retrieved February 1, 2020 from https://www.digitaladventures.com/news/2016/5/2/how-learning-to-code-can-help-develop-a-growth-mindset [Cited on page 174]

Kristin Villanueva. 2018a. *How Having a Growth Mindset Can Help You Learn to Code*. Retrieved February 1, 2020 from http://blog.mindsetworks.com/entry/how-having-a-growth-mindset-can-help-you-learn-to-code [Cited on page 174]

Kristin Villanueva. 2018b. *How Learning to Code Can Help You Develop a Growth Mindset*. Retrieved February 1, 2020 from http://blog.mindsetworks.com/entry/how-learning-to-code-can-help-you-develop-a-growth-mindset [Cited on page 174]

Joke Voogt, Petra Fisser, Jon Good, Punya Mishra, and Aman Yadav. 2015. Computational thinking in compulsory education: Towards an agenda for research and practice. *Education and Information Technologies* 20, 4 (01 Dec. 2015), 715–728. https://doi.org/10.1007/s10639-015-9412-6 [Cited on page 33]

Lev Vygotsky. 1978. *Mind in Society*. Harvard University Press, London. [Cited on page 43]

Michael Weigend. 2019. Computer Science Unplugged and the Benefits of Computational Thinking. *Constructivist Foundations* 14, 3 (2019), 352–353. https://constructivist.info/14/3/352.weigend [Cited on page 120]

David Weintrop, Elham Beheshti, Michael Horn, Kai Orton, Kemi Jona, Laura Trouille, and Uri Wilensky. 2016. Defining Computational Thinking for Mathematics and Science Classrooms. *Journal of Science Education and Technology* 25, 1 (01 Feb. 2016), 127–147. https://doi.org/10.1007/s10956-015-9581-5 [Cited on page 205]

David Weintrop and Uri Wilensky. 2015. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-Based Programming. In *Proceedings of the 14th International Conference on Interaction Design and Children* (Boston, Massachusetts) *(IDC '15)*. Association for Computing Machinery, New York, NY, USA, 199–208. https://doi.org/10.1145/2771839.2771860 [Cited on page 112]

David Weintrop and Uri Wilensky. 2019. Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers & Education* 142 (Dec. 2019), 103646. https://doi.org/10.1016/j.compedu.2019.103646 [Cited on pages 57 and 112]

Gordon Wells. 1999. *Dialogic inquiry: Towards a socio-cultural practice and theory of education*. Cambridge University Press. [Cited on page 119]

Allan Wigfield, Jacquelynne S. Eccles, Douglas Mac Iver, David A. Reuman, and Carol Midgley. 1991. Transitions during early adolescence: Changes in children's domain-specific self-perceptions and general self-esteem across the transition to junior high school. *Developmental psychology* 27, 4 (1991), 552. [Cited on page 183]

Jeannette M. Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (March 2006), 33. https://doi.org/10.1145/1118178.1118215 [Cited on pages 2, 27, 28, 36, 53, 71, 79, and 80]

Jeannette M. Wing. 2008. Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366, 1881 (2008), 3717–3725. https://doi.org/10.1098/rsta.2008.0118 [Cited on pages 28, 29, 31, and 82]

Jeannette M. Wing. 2011. Research Notebook: Computational Thinking–What and Why? *The Link Magazine* (2011). https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why [Cited on pages 28, 29, and 31]

Eric B. Winsberg. 2010. *Science in the age of computer simulation*. University of Chicago Press, Chicago. [Cited on page 79]

Niklaus Wirth. 1993. Recollections about the Development of Pascal. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA) *(HOPL-II)*. Association for Computing Machinery, New York, NY, USA, 333–342. https://doi.org/10.1145/154766.155378 [Cited on page 107]

David Wood, Jerome S. Bruner, and Gail Ross. 1976. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry* 17, 2 (April 1976), 89–100. https://doi.org/10.1111/j.1469-7610.1976.tb00381.x [Cited on pages 43 and 119]

Aman Yadav, Ninger Zhou, Chris Mayfield, Susanne Hambrusch, and John T. Korb. 2011a. Introducing Computational Thinking in Education Courses. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. ACM, New York, NY, USA, 465–470. https://doi.org/10.1145/1953163.1953297 [Cited on pages 147 and 153]

Aman Yadav, Ninger Zhou, Chris Mayfield, Susanne Hambrusch, and John T. Korb. 2011b. Introducing Computational Thinking in Education Courses. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. Association for Computing Machinery, New York, NY, USA, 465–470. https://doi.org/10.1145/1953163.1953297 [Cited on pages 147 and 153]

David S. Yeager, Paul Hanselman, Gregory M. Walton, Jared S. Murray, Robert Crosnoe, Chandra Muller, Elizabeth Tipton, Barbara Schneider, Chris S. Hulleman, Cintia P. Hinojosa, David Paunesku, Carissa Romero, Kate Flint, Alice Roberts, Jill Trott, Ronaldo Iachan, Jenny Buontempo, Sophia Man Yang, Carlos M. Carvalho, P. Richard Hahn, Maithreyi Gopalan, Pratik Mhatre, Ronald Ferguson, Angela L. Duckworth, and Carol S. Dweck. 2019. A national experiment reveals where a growth mindset improves achievement. *Nature* 573, 7774 (Aug. 2019), 364–369. https://doi.org/10.1038/s41586-019-1466-y [Cited on page 64]

David S. Yeager, Carissa Romero, Dave Paunesku, Christopher S. Hulleman, Barbara Schneider, Cintia Hinojosa, Hae Yeon Lee, Joseph O'Brien, Kate Flint, Alice Roberts, Jill Trott, Daniel Greene, Gregory M. Walton, and Carol S. Dweck. 2016. Using design thinking to improve psychological interventions: The case of the growth mindset during the transition to high school. *Journal of Educational Psychology* 108, 3 (2016), 374–391. https://doi.org/10.1037/edu0000098 [Cited on page 64]

David S. Yeager and Gregory M. Walton. 2011. Social-Psychological Interventions in Education. *Review of Educational Research* 81, 2 (June 2011), 267–301. https://doi.org/10.3102/0034654311405999 [Cited on page 186]

Jacob W. A. Young. 1906. *The teaching of mathematics in the elementary and the secondary school*. Longmans, Green and Company. [Cited on page 53]

Acknowledgments

Grazie

A Simone, per aver creduto in me, per avermi guidato in modo fermo ma discreto. Sei un grande modello per me.
A Renzo e Rebecca, per le avventure didattico-computazionali.
To the whole Department of Fun Stuff, and especially to Tim, for hosting me and guiding me during an incredible Kiwi adventure.
To Valentina and Michael, for giving valuable comments on the first draft of this work.
Ad Enrico, Luca, Isabella, alle Aladdine e agli Aladdini, a tutto il comitato delle OPS, e a tutta la comunità del CINI, perché non smettiamo di crederci.
Ad Olivia e a tutto il gruppo di Didattica delle STEM, per gli incredibili viaggi epistemologici.
Ad Elena, per le tesi sul "coding".
A Davide e Ugo, per aver vigilato in background.
A tutti gli altri docenti e ricercatori con cui ho avuto il piacere di collaborare.
Ad Allegra e a Stefano, a tutti i colleghi del XXXII, e a tutti gli occupanti dell'Underground, per il percorso insieme.
A Marco, per le tante avventure che ci aspettano nella "computing" education.
A tutti gli insegnanti e a tutti i docenti che hanno partecipato agli studi, e ad Alessandra e a Rita, perché siete uniche.
A Carmelo, Angela e a tutti i mentor del CoderDojo Bologna, da dove tutto è partito.

Alla mia mamma e al mio papà, per avermi fatto diventare chi sono, ma sempre permettendomi di decidere chi volevo essere.
Alla mia tata & family, nonna e zii, perché sapete che vi voglio bene anche se non ci vediamo spesso.

Al gruppo dei "soldi pubblici": Andrea, Andrea, Andrea, Laura, Alessandro, Giacomo, perché mi regalate gioia e risate dopo lunghi periodi di duro lavoro.

Ma soprattutto grazie a Chiara, la mia patata, perché mi ricordi di guardare in alto. Voglio crescere insieme a te.